UiO : University of Oslo

James D. Trotter

High-performance finite element computations

Performance modelling, optimisation, GPU acceleration & automated code generation

Thesis submitted for the degree of Philosophiae Doctor

Department of Informatics
Faculty of Mathematics and Natural Sciences

Simula Research Laboratory

2020

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 2342

ISSN 1501-7710

Cover: Hanne Baadsgaard Utigard.
Print production: Reprosentralen, University of Oslo.

Preface

This thesis is submitted in partial fulfilment of the requirements for the degree of *Philosophiae Doctor* at the University of Oslo. The research presented here was conducted at Simula Research Laboratory, under the supervision of Professor Xing Cai, Dr. Johannes Langguth, and Dr. Simon W. Funke. This work was supported by the Norwegian Research Council through grant 251186.

The thesis is made up of three research papers, presented in chronological order of writing. The underlying, common theme is the use of recent computing hardware to accelerate certain calculations pertaining to a class of numerical methods known as finite element methods. The papers are preceded by an introductory chapter that relates them together and provides background information and motivation for the work. The first and third papers are joint work with Xing Cai and Johannes Langguth, whereas the second paper is written together with Xing Cai and Simon W. Funke.

Acknowledgements

My sincerest thanks to Xing Cai for, in my opinion, going above and beyond on many occasions. I feel very fortunate to have had such a collaboration and grateful that you have shared so much of your knowledge. Thank you for being kind, trusting and generous.

I also thank Johannes Langguth, especially for guiding me during the early stages of this project, but also for the many fun and varied discussions we have had throughout. I want to thank Simon W. Funke for his involvement at important moments, and whose encouragement meant a lot.

I have many colleagues to thank for helping in different ways, including Kristian Gregorius Hustad, Hermenegild Arevalo, Alban Souche and Valeriya Naumova for assisting with data or hardware that was used during my work. Tore H. Larsen deserves a special mention for his tireless commitment in supporting the eX3 platform and its users. In addition, I thank Martin Sandve Alnæs, Jørgen Dokken, Chad Jarvis, Miroslav Kuchta and Andreas Thune for helpful discussions along the way.

Simula as a whole deserves special thanks for firmly placing everyone's health and well-being above all else in trying times.

Finally, I thank my loving wife, family and friends.

James D. Trotter
Oslo, September 2020

List of Papers

Paper I

Trotter, J. D., J. Langguth and X. Cai (2020). "Cache simulation for irregular memory traffic on multi-core CPUs: Case study on performance models for sparse matrix-vector multiplication". In: *Journal of Parallel and Distributed Computing*, *144*, pp. 189–205. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2020.05.020.

Paper II

Trotter, J. D., X. Cai and S. W. Funke. "On memory traffic and optimisations for low-order finite element assembly algorithms on multi-core CPUs".
Submitted for publication.

Paper III

Trotter, J. D., J. Langguth and X. Cai. "Leveraging GPU-accelerated finite element computation with automated code generation: A holistic approach".
Submitted for publication.

Contents

List of Figures

List of Tables

Introduction

Paper I

Paper II

Paper III

List of Algorithms

Chapter 1

Introduction

Mathematical modelling and numerical simulation are staples of many scientific and engineering disciplines today. Models based on partial differential equations (PDEs), in particular, are extraordinarily wide-ranging. These models and their accompanying numerical simulations are used, for example, to forecast the weather as well as to inform researchers and clinicians about mechanisms underlying cardiac disease and neurological disorders.

The broad topic of this thesis relates to finite element methods, a popular class of numerical methods for solving PDEs. More specifically, the work presented here contributes to solving such problems faster with the use of recent computing hardware. This is done by pursuing three main ideas while analysing and optimising the performance of some of the central calculations that are involved.

First, we thoroughly investigate how memory traffic impacts finite element computations, especially the assembly of linear systems. The overall operation of finite element solvers requires both assembling and solving linear systems of equations, and the former often contributes a substantial portion of the total computations to be carried out. Our work goes well beyond analysing performance solely in terms of counting the number of arithmetic operations that are performed. As a result, we uncover some highly effective optimisation strategies, making more efficient use of the memory subsystems on shared memory, multi-core CPUs.

In addition to assembly, we take a close look at memory traffic resulting from sparse matrix-vector multiplication (SpMV) kernels. One of the many applications of such kernels is for solving sparse linear systems that arise from finite element methods. The significance of memory traffic is already well established in this case. But the irregular memory access patterns that are intrinsic to SpMV kernels still make it difficult to pinpoint bottlenecks and predict their performance in a precise manner. In light of this, we contribute a quantitative model that can be used to more accurately understand SpMV performance on multi-core CPUs, even for highly irregular matrices.

Second, we explore how to accelerate finite element solvers by offloading major calculations, particularly the assembly of linear systems, to graphics processing units, or GPUs. Large parts of the scientific computing community have already embraced a paradigm of heterogeneous computing, where performance-critical computations are offloaded to accelerator hardware. In most cases, these accelerators come in the form of GPUs. Driven by demands for higher performance and better energy efficiency, accelerators are now vital to large-scale computations on many current supercomputers. In all likelihood, heterogeneity will continue to be an essential feature of future computing systems.

The third and final point is not only to optimise and accelerate finite element solvers, but at the same time also to build on established tools for automated code

generation. In doing so, we aim to bolster the productivity of domain scientists whose work relies on finite element methods. Proficiency in implementing the intricate details of high-performance finite element solvers should not be needed to make good use of them for numerical simulations.

The main part of this thesis is made up of three research papers that examine the various aspects mentioned above.

Outline

The rest of this chapter proceeds with Section 1.1 giving some brief, technical background to support the rest of the work in this thesis. Thereafter, we present our main research questions in Section 1.2, before Section 1.3 summarises the three research papers that make up the main part of the thesis. A discussion and our main conclusions follow in Section 1.4. Finally, the three research papers are then presented in their entirety.

1.1 Background

In this section, we recount some central background material that is needed to later thoroughly summarise our research. Section 1.1.1 first reviews some of the main points involved in finite element computations. Next, we give a short description in Section 1.1.2 of typical multi-core CPU architecture, emphasising the memory subsystem and how it impacts performance of memory-intensive applications. This is followed by a brief account of matters related to GPU computing in Section 1.1.3. Finally, we give some background on automated code generation in Section 1.1.4.

1.1.1 Finite element methods

This section provides a cursory overview of computational procedures that are involved when using finite element methods to solve PDEs. This is done through the classical example of solving Poisson's equation. More in-depth accounts of various aspects surrounding finite element methods can be found in a number of textbooks, such as Axelsson and Barker [2001], Ciarlet [2002], and Ern and Guermond [2004].

Poisson's equation. To begin, we state our example PDE problem for a polygonal domain $\Omega \subset \mathbf{R}^d$, whose boundary $\partial\Omega$ consists of two disjoint parts, Γ_D and Γ_N. Then, Poisson's equation with a diffusion coefficient $\kappa > 0$, source term f, boundary term g, and mixed Dirichlet-Neumann boundary conditions is given by

$$\begin{aligned} -\kappa\nabla^2 u &= f \quad && \text{in } \Omega, \\ u &= 0 && \text{on } \Gamma_D, \\ \nabla u \cdot \mathbf{n} &= g && \text{on } \Gamma_N, \end{aligned} \tag{1.1}$$

where $\mathbf{n}$ denotes the outward-pointing unit normal on the boundary.

Through a standard procedure, a finite element method translates the above problem into a system of linear equations. As a rough outline, the computations to be

performed consist of two essential stages: Assembling the linear system and solving it. These are described in further detail below together with some prerequisite steps that are also needed.

The basic approach remains largely the same even for a time-dependent, non-linear, system of PDEs. However, in this case, a sequence of linear systems arises. These linear systems are successively assembled and solved by means of the same fundamental computations as before.

Weak formulation. The first step is to introduce a weak form of the PDE problem. In the case of Poisson's equation, as stated in Eq. (1.1), the standard weak formulation consists of finite-dimensional test and trial spaces, denoted by V and U, respectively, as well as a bilinear form $a\colon V \times U \to \mathbf{R}$ and a linear form $L\colon V \to \mathbf{R}$. These are given by

$$a(v,u) = \int_\Omega \kappa \nabla v \cdot \nabla u \,\mathrm{d}x, \quad \text{and} \quad L(v) = \int_\Omega fv \,\mathrm{d}x + \int_{\Gamma_N} gv \,\mathrm{d}s. \tag{1.2}$$

In our case, the test and trial spaces are subspaces of the Sobolev space $H^1(\Omega)$ with the additional requirement that functions must vanish on the Dirichlet boundary Γ_D. In the following, we assume that the test and trial spaces are the same, $U = V$.

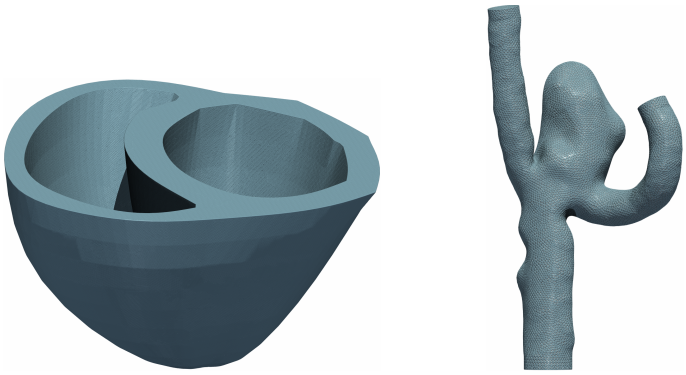
The goal is now to find a solution $u \in U$, belonging to the trial space, such that $a(v,u) = L(v)$ for every test function $v \in V$. If $\phi_0, \phi_1, \ldots, \phi_{N-1} \in U$ is a basis for the trial space, then a solution can be expanded into a sum, $u = \sum_{j=0}^{N-1} x_j \phi_j$. The coefficients, $x_0, x_1, \ldots, x_{N-1} \in \mathbf{R}$, are often referred to as degrees of freedom.

Furthermore, if $\psi_0, \psi_1, \ldots, \psi_{N-1} \in V$ is a basis for the test space, then the problem of finding a solution u may be formulated as a linear system of equations, $Ax = b$. The coefficient matrix is $A_{i,j} = a(\psi_i, \phi_j)$ and the right-hand side vector is $b_i = L(\psi_i)$, for $0 \leq i, j < N$. Also, the values $x_0, x_1, \ldots, x_{N-1}$ resulting from solving this linear system are precisely the coefficients of the trial function u with respect to the trial space basis. Solving the linear system therefore yields a solution to the variational formulation of the PDE.

Finite elements. Next, a computational mesh is needed to represent the problem domain and to allow the integrals that appear in the variational formulation to be evaluated more easily. Thus, the domain is partitioned into cells, usually triangles, tetrahedra, or other convex polytopes. It is customary to require that the intersection of any two cells is either empty or a vertex, edge or face belonging to both cells. Figure 1.1 shows two examples of unstructured, tetrahedral meshes that are used again later in this thesis.

As a rule, an unstructured mesh is used, meaning that there is no simple or obvious pattern that describes the connectivity between mesh cells. An explicit representation of the mesh cell connectivity is therefore needed. During mesh-based computations, such as assembling and solving linear systems obtained from finite element methods, this becomes a major source of irregular memory traffic and a serious impediment to performance.

Once a mesh is given, each mesh cell is associated with a local finite element space, spanned by a set of carefully chosen basis functions, often referred to as shape



(a) *Cardiac mesh* **(b)** *Aneurysm mesh*

Figure 1.1. Unstructured, tetrahedral meshes from cardiac modelling [Marciniak et al. 2017] and blood flow simulations [Aneurisk-Team 2012].

functions. The most common example is that of Lagrange elements, where the shape functions are Lagrange interpolating polynomials with respect to a specific set of points, or nodes, of the cell. Besides Lagrange elements, there exists a whole range of other, widely used function spaces and finite elements [Arnold and Logg 2014].

In any case, local shape functions on each mesh cell are combined to form a basis for a global finite element space, consisting of polynomials that are defined piecewise with respect to the computational mesh. Ultimately, such global finite element spaces, accompanied by their deliberately constructed bases, appear as test or trial spaces in weak formulations of PDEs, such as Eq. (1.2).

Finite element assembly. After introducing a computational mesh and suitable finite elements, the work comes down to assembling and solving linear systems of equations. A linear system, $Ax = b$, is assembled from problem-specific element matrices A_T and vectors b_T on each mesh cell T.

In the case of the variational forms in Eq. (1.2), the element matrices and vectors are

$$(A_T)_{i,j} = \int_T \kappa \nabla \psi_i^T \cdot \nabla \phi_j^T \, \mathrm{d}x, \quad \text{and} \quad (b_T)_i = \int_T f \psi_i^T \, \mathrm{d}x + \int_{\partial T \cap \Gamma_N} g \psi_i^T \, \mathrm{d}s, \tag{1.3}$$

where ψ_i^T and ϕ_j^T are shape functions belonging to local test and trial spaces on the mesh cell T, respectively.

The global matrix, $A \in \mathbf{R}^{N,N}$, of the linear system is computed as a sum over the mesh cells,

$$A = \sum_{T \in \mathcal{T}} P_T A_T Q_T^{\mathsf{T}}, \tag{1.4}$$

where $A_T \in \mathbf{R}^{n,n}$ is an element matrix for the mesh cell T. The matrices $P_T \in \mathbf{R}^{N,n}$ and $Q_T \in \mathbf{R}^{N,n}$ are designed to scatter contributions from the element matrix to the correct

locations of the global matrix A. Without going into too much detail, P_T and Q_T are defined in terms of local-to-global mappings from shape functions on each mesh cell to the global test and trial spaces, respectively.

The right-hand side vector is assembled in a similar manner.

To assemble linear systems as described above, some implementation of certain problem-specific computational kernels is needed. With respect to our current example, these kernels must evaluate the integrals in Eq. (1.3), when they are provided with a given mesh cell T. In general, hand-written kernels are often provided for each individual problem. Alternatively, such kernels can be produced by automatically generating code from a suitable high-level description of the finite element problem at hand, as described in Section 1.1.4.

One final issue related to assembly is that boundary conditions must be taken into account. Neumann conditions are readily incorporated into the variational form, for example, in the linear form of Eq. (1.2) and element vector of Eq. (1.3). Dirichlet boundary conditions, on the other hand, are accounted for by modifying the linear system during or after it has been assembled. First, entries of an element matrix are set to zero if their rows or columns correspond to degrees of freedom that are subject to Dirichlet boundary conditions. This is done prior to adding an element matrix to the global matrix. Second, after assembling the global matrix, its diagonal entries are set to one for rows where Dirichlet boundary conditions apply. The corresponding entries of the right-hand side vector are modified according to the Dirichlet boundary values prescribed.

Iterative linear solvers. Linear systems of equations derived from finite element methods are sparse and often very large. As such, appropriate linear solvers are needed. The work in this thesis relies on standard, iterative methods [Saad 2003], for example, the conjugate gradient method, which is used in Paper III.

A crucial, often performance-critical part of such methods is repeatedly carrying out sparse matrix-vector multiplications (SpMV). These operations inherently lead to irregular memory traffic and performance that depends heavily on the memory subsystem of the underlying hardware rather than its floating-point capabilities.

Sparse matrices are most commonly encountered in the compressed sparse row (CSR) format. A standard, shared memory-parallel SpMV kernel using the CSR format is the main example studied in connection with performance modelling of irregular, bandwidth-limited computations in Paper I. The CSR format also appears in Papers II and III, due to being commonly used for global, sparse matrices that are assembled during finite element computations.

To briefly explain, consider a sparse matrix with M rows, N columns and K non-zero entries. The matrix is then stored as K non-zero values, $a_0, a_1, \ldots, a_{K-1}$, together with a compressed representation of the location of each non-zero. More specifically, non-zero locations are stored as K column indices $j_0, j_1, \ldots, j_{K-1}$ and $(M + 1)$ row pointers $r_0, r_1, \ldots, r_M$. The non-zeros are sorted according to the row they belong to, such that r_i and $(r_{i+1} - 1)$ are the indices of the first and last non-zeros of the i-th row, respectively. In other words, a non-zero a_k belongs to the i-th row if $r_i \leq k < r_{i+1}$. For further details, see, for example, Saad [2003].

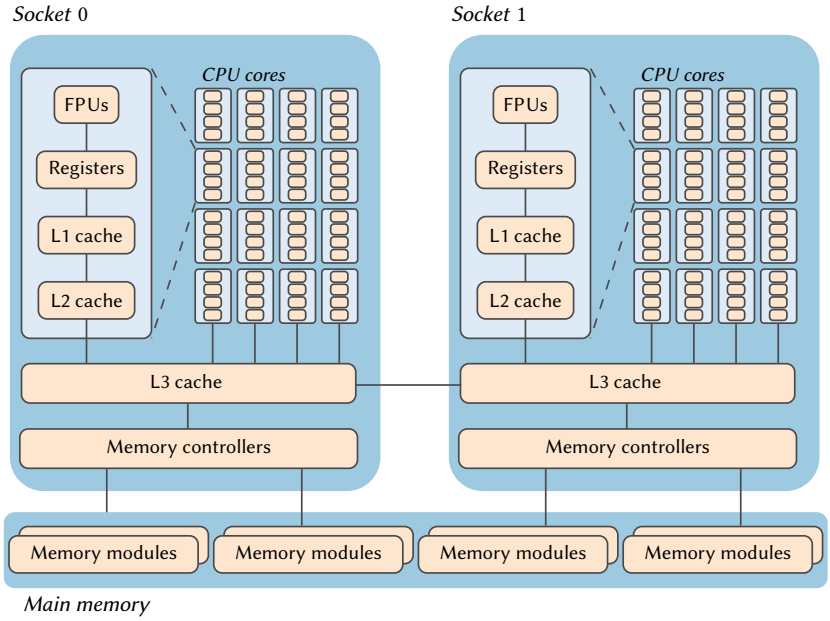


Figure 1.2. Memory hierarchy of a dual-socket, multi-core CPU.

1.1.2 Multi-core CPUs

An important part of this thesis is centred on memory-related optimisations for finite element codes. For this reason, we give a high-level description of the architecture and memory subsystem of a typical multi-core CPU. Further details can be found in appropriate documentation for specific multi-core CPUs, such as Intel Xeon [Intel Corporation 2018], AMD Epyc [Guo 2019], or ARM64 [Guo et al. 2019].

Common multi-core CPU systems are made up of one or more sockets, each containing up to a few dozen CPU cores, as illustrated in Figure 1.2. CPU cores contain floating-point units (FPUs) for carrying out floating-point calculations. These calculations can only operate on data that lies in a CPU core's registers. In other words, registers are the working memory of a CPU.

Vectorisation is often used to perform calculations in parallel on multiple values stored consecutively in a single register. Usually, the vector width is 2, 4, 8, or 16, signifying the number of values that fit in a single register and may thus be handled in parallel.

Data that is needed for computation, but is not already located in registers, must be fetched from memory. Conversely, intermediate results that do not fit in registers, or results that are otherwise to be saved for later, must be written to memory. Moreover, memory is organised in a multi-level hierarchy. In addition to registers, each core has its own first- and second-level cache. These are fast memories used for recently accessed or soon to be used data. A larger, third-level cache is most often shared by all the cores of a socket. Finally, each socket has one or more memory controllers that

are used to transfer data between third-level caches and main memory.

Caches are in reality very complex, and many of the details surrounding their exact implementation are never revealed by CPU vendors. For practical work, such as designing cache-friendly or cache-oblivious algorithms, it is much more useful to rely on simplified models, such as the ideal cache model described by Frigo et al. [2012]. The cache simulation methodology in Paper I is partly based on this model.

Many memory-intensive codes are bandwidth-limited in the sense that their performance is chiefly a result of how much data can be moved to and from memory at any given time. A theoretical upper limit on memory bandwidth is obtained by multiplying the transfer rate of memory modules by the number of channels used by the memory controllers. For example, a system with DDR4 memory operating at 2 666 MHz and six memory channels has a theoretical bandwidth of about 128 GB/s.

However, for various reasons, the theoretical bandwidth does not reflect what may be achieved in reality. Instead, a rudimentary, but effective way to gauge the achievable performance is through a microbenchmark, such as STREAM [McCalpin 1995]. This benchmark is routinely used to measure the practically achievable memory bandwidth of multi-core CPUs in the case of streaming memory access patterns. It is, for example, used in Papers I and II.

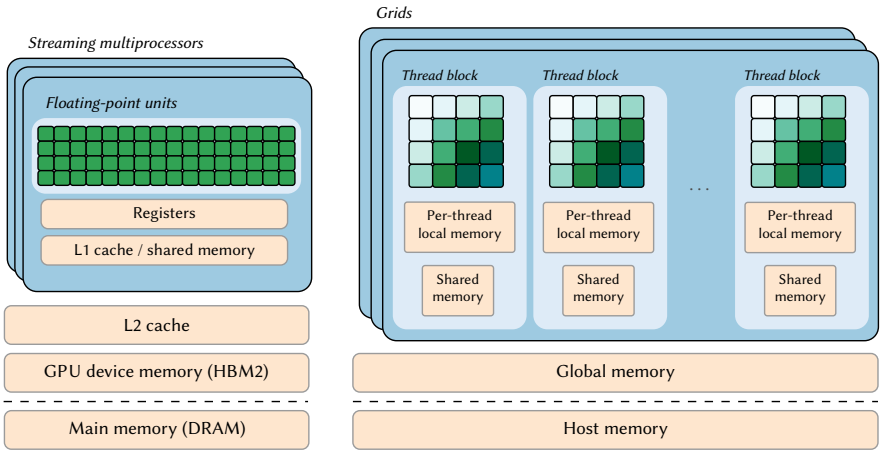
Finally, we note that multi-core CPUs employ various measures to maintain correct results whenever several CPU cores operate on data from the same location in memory. Cache coherency ensures that data residing in one CPU core's cache is up-to-date, in the event that the same data is modified by another CPU core. Even so, it is often necessary to perform some form of explicit synchronisation between threads working on the same data. In this case, atomic operations are regularly used to ensure that a single CPU core updates and writes a certain piece of data to memory without interference from others.

For example, a shared memory parallel implementation of the standard, cellwise finite element assembly algorithm (see Section 1.1.1) can use atomic operations to prevent race conditions between different threads updating the same global matrix values. Failing to do so can lead to incorrect results. This approach is encountered in Papers II and III.

1.1.3 GPU computing

Researchers have used GPUs for more than a decade to solve problems in scientific computing [Che et al. 2008; Owens et al. 2007]. In 2006, NVIDIA announced its Tesla GPU architecture together with the CUDA programming model, designed to make general-purpose computations on NVIDIA GPUs more accessible [Kirk and Hwu 2010]. OpenCL [Khronos OpenCL Working Group 2020] is another GPU computing standard that may be used with GPUs from several vendors. The GPU-related work in this thesis is nonetheless based on CUDA, which still remains the most commonly used method of GPU programming.

In this section, we briefly describe some of the main features of the CUDA programming model and NVIDIA's GPU architecture. More often than not, a detailed understanding of both is needed to properly benefit from GPU acceleration. Further details are found in the *CUDA C Programming Guide* [NVIDIA Corporation 2020b].



(a) *GPU architecture* **(b)** *CUDA memory hierarchy*

Figure 1.3. GPU architecture and logical memory hierarchy in the CUDA programming model.

CUDA programming model. CUDA is a heterogeneous programming model where a program executing on a host processor can offload parallel tasks to a separate *CUDA device*, which, in practice, is just another name for an NVIDIA GPU. The programmer decomposes a parallel task into a large number of fine-grained tasks, each of which is to be executed by a thread.

The programmer also arranges these fine-grained tasks so that threads map onto a three-dimensional *grid* made up of three-dimensional *thread blocks*. Each thread block consists of a limited number of threads that may cooperate and synchronise, though the order of execution among thread blocks is not specified.

Another feature of the way parallel tasks are offloaded is that each thread executes the same program, called a *kernel*. Individual threads can still operate on different data or execute different instructions by having the kernel depend on a thread's location in the grid. Kernels are written in C++ or Fortran and must be compiled to PTX assembly language code or GPU-specific binary code before they can be launched from the host. There exists a large collection of libraries containing optimised CUDA kernels for different applications, including linear solvers [Naumov et al. 2015; NVIDIA Corporation 2020c; Tomov et al. 2010].

NVIDIA GPU architecture. A typical NVIDIA GPU consists of several *Streaming Multiprocessors (SMs)*, as shown in Figure 1.3(a). Each SM has a large number of floating-point units (FPUs) and a warp scheduler to schedule threads for execution. In addition, each SM has a limited amount of registers as well as a fast memory that is divided between L1 cache and shared memory. The latter acts as a form of user-managed cache or scratchpad memory. Beyond that, SMs share a single L2 cache and the GPU device memory.

As an example of a recent GPU model, NVIDIA V100 [NVIDIA Corporation 2017] consists of 80 SMs based on the Volta microarchitecture. Each SM features 64 and 32 single- and double-precision FPUs, respectively. Consequently, the peak double-precision performance is 7800 Gflop/s. There are also a number of so-called tensor cores for accelerating specialised operations on small matrices. Furthermore, SMs carry 256 kB of register memory and 128 kB of L1 cache, where up to 96 kB can be set aside for shared memory. An L2 cache of 6144 kB is shared among SMs. Device memory consists of 32 GB of high-bandwidth memory (HBM2) with a peak bandwidth of 900 GB/s.

When a kernel is launched on a GPU, different thread blocks are assigned to different SMs to be executed concurrently. Furthermore, at each instruction issue, an SM will select a small group of thirty-two consecutive threads from the same thread block, called a *warp*, to be scheduled for execution. An SM can schedule a new warp each cycle, provided that one is available for execution. The entire warp generally executes one common instruction at a time. However, each thread has its own execution context, including program counter and register state, so it can operate on its own data or diverge from other threads by not participating in the execution of a particular instruction. On the other hand, thread divergence can adversely impact performance and should be avoided, if possible.

Performance considerations. Finally, we mention some recurring themes in optimising CUDA kernels for NVIDIA GPUs. All of these relate somehow to keeping floating-point units busy by quickly and efficiently supplying data that is needed for computation. The points below are all relevant to various degrees with respect to the GPU-acceleration studied in Paper III. More information on how to obtain good performance for GPU computations is found in the *CUDA C++ Best Practices Guide* [NVIDIA Corporation 2020a].

The logical memory hierarchy presented to the programmer by the CUDA programming model is depicted in Figure 1.3(b). Each thread may access its *local memory*, which is private to the thread. There is also a *shared memory*, which is available to a whole thread block, and *global memory*, which is shared by all threads on a device. Finally, the global memory of a CUDA device is separate from the host's memory, so data must be explicitly transferred to or from the device.

L2 cache and GPU device memory are used to serve global memory accesses. Data in shared memory is guaranteed to reside in an SM's fast, on-chip memory, but per-thread local memory may reside in registers, L1 cache, L2 cache, or even GPU device memory.

Device memory may offer a high bandwidth, but the latency cost associated with individual memory accesses is also very high. It is therefore important to maintain high occupancy. That is, each SM should at all times have enough active warps ready to be scheduled for execution. The number of warps simultaneously handled by an SM is often limited by the availability of registers and shared memory. Thus, one should at the same time attempt to limit the amount of registers and shared memory required by each thread block.

On a related note, the concept of *coalesced memory access* is needed to achieve the

highest possible throughput for accesses from global memory. This occurs whenever threads in a warp access consecutive memory locations in global memory. It is often worthwhile to rearrange data specifically to accomplish coalescing and thus improve throughput for global memory accesses.

Finally, transferring data between a host CPU's main memory and GPU device memory is typically done over a PCIe connection with a meagre bandwidth of 16 GB/s. As a consequence, such transfers should be kept at a minimum to prevent them from becoming a bottleneck.

1.1.4 Automated code generation

Reaching high performance in scientific codes is often a matter of considerable effort, requiring proficiency in parallel programming and performance optimisation. If GPUs are used, then knowledge of special-purpose programming models may be needed too. At worst, high-performance tools for numerical simulations are simply unobtainable for researchers and others from domains outside of high-performance computing. Many could surely benefit from such tools, but they may not have the time or skills needed to develop them.

For some problems in scientific computing, automated code generation has provided a solution that combines high performance with productivity and accessibility. In some sense, this is a continuation of auto-tuning strategies used, for example, in numerical linear algebra [Clint Whaley et al. 2001; Vuduc et al. 2005] and stencil codes [Datta et al. 2008]. Automated code generation has, in fact, received a good deal of attention in the area of stencil codes [Christen et al. 2011; Han et al. 2011; Holewinski et al. 2012; Sourouri et al. 2017; Tang et al. 2011; Unat et al. 2011; Unat et al. 2012; Zhang and Mueller 2012]. It has also been used to generate GPU code for array-based computations with tools such as PyCUDA and PyOpenCL [Klöckner et al. 2012], and Loo.Py [Klöckner 2014; Klöckner et al. 2016].

For PDE solvers based on finite element methods, open-source frameworks such as FEniCS [Logg et al. 2012] and Firedrake [Rathgeber et al. 2016] use automated code generation techniques based on the Unified Form Language (UFL) [Alnæs et al. 2014] to bring high-performance finite element computations to a wider audience.

Recall the variational forms for Poisson's equation in Eq. (1.2), and the corresponding expressions for calculating element matrices and vectors in Eq. (1.3). Using FEniCS, the appropriate computational kernels for computing such element vectors and matrices are automatically generated from the UFL code shown in Algorithm 1.1. In this example, first-order Lagrange elements on tetrahedral mesh cells are used. The variational forms are spelled out with the help of volume and area elements **`dx`** and **`ds`** and operators such as **`inner`** and **`grad`**, which represent inner product and gradient operators, respectively.

UFL deliberately mimics standard notation for finite element methods, making it easier for users to translate finite element problems into code. A more detailed guide to using FEniCS and UFL to solve PDEs is found in the FEniCS tutorial [Langtangen and Logg 2017].

```
element = FiniteElement("Lagrange", tetrahedron, 1)
coords = VectorElement("Lagrange", tetrahedron, 1)
mesh = Mesh(coords)

V = FunctionSpace(mesh, element)
u = TrialFunction(V)
v = TestFunction(V)
f = Coefficient(V)
g = Coefficient(V)
kappa = Constant(mesh)

a = kappa * inner(grad(u), grad(v)) * dx
L = inner(f, v) * dx + inner(g, v) * ds
```

Algorithm 1.1. UFL code for Poisson's equation with mixed Dirichlet-Neumann boundary conditions.

1.2 Research questions

In this section, we present a series of questions that guide the research undertaken in this thesis. Our questions relate to the subjects of high-performance computing and finite element methods. Since these are vast topics, we mention a few ways in which we limit the scope of our research.

First, we consider only some of the most commonly used finite element methods, based on low-order piecewise polynomial basis functions. Linear and quadratic Lagrange elements, for example, are widely used in practice and are therefore important to many applications. In other words, we do not consider higher-order and spectral methods, which usually require considerably different algorithms to implement efficiently.

Further, in connection with linear solvers, we only concern ourselves with iterative solvers for sparse linear systems, rather than direct solvers. The former are often used in connection with large-scale problems based on unstructured meshes. Although iterative solvers usually incorporate a preconditioning step, we do not touch on the topic of preconditioning.

A related point is that our study of finite element solvers is based around performing global matrix assembly. Matrix-free and related methods are therefore not considered. These are known to be mostly advantageous for higher-order methods [Cantwell et al. 2011; Kronbichler and Kormann 2012; Reguly and Giles 2015; Vos et al. 2010].

Several of the following questions originate from an overarching concern of finding memory-related factors that influence the performance of finite element solvers. Recall that finite element computations can be broken down into two stages: assembling and solving linear systems. Hence, we are naturally led to consider memory-related performance issues of each stage.

Memory traffic has so far received little attention in the context of finite element assembly, even though such computations heavily feature irregular memory accesses. Thus, our first research question concerns memory-related performance issues of finite element assembly.

Question 1. How do irregular memory accesses influence the performance of finite

element assembly procedures and how should such algorithms be implemented to mitigate memory-related performance issues?

Paper II is primarily concerned with the above question for the implementation of finite element assembly on shared memory, multi-core CPUs. The same issues are also dealt with in Paper III when offloading finite element assembly to a GPU.

For iterative linear solvers, on the other hand, the connection between memory traffic and performance is already well recognised. Since they are so widely applicable, also outside the domain of finite elements and PDEs, it is fair to say that iterative solvers have been more thoroughly studied in general.

In particular, sparse matrix-vector multiplication (SpMV) is one of the cornerstones of iterative solvers for sparse linear systems, where such operations are carried out repeatedly and often dominate performance. Although SpMV kernels are mostly limited by available memory bandwidth, irregular memory accesses often make it difficult to predict their performance accurately.

Our next research question aims to investigate SpMV performance in more detail. Earlier work by Langguth et al. [2015a] develops a performance model for SpMV on both multi-core CPUs and GPUs for a special case of matrices that arise from a cell-centred finite volume method on tetrahedral meshes. Partly as a continuation of this effort, we ask about SpMV performance for general, sparse matrices. But we consider only multi-core CPUs to limit the scope of our research.

Question 2. What is needed to quantitatively understand the performance of parallel sparse matrix-vector multiplication on multi-core CPUs?

This question is pursued in Paper I, where a detailed quantitative performance model is developed. As it happens, the methodology developed in the paper is not specifically tied to SpMV. On the contrary, it is much more general, in the sense that it is relevant for other irregular, bandwidth-limited kernels too.

The remaining questions relate to offloading finite element assembly to GPUs, a topic mainly addressed in Paper III. Many aspects of GPU-accelerated finite element solvers have been carefully investigated in the past, including GPU-based global assembly [Cecka et al. 2010; Reguly and Giles 2015] and GPU-based iterative linear solvers [Anzt et al. 2020; Anzt et al. 2017; Naumov et al. 2015].

Our attention is devoted mostly to GPU-based assembly of linear systems, where the topic of memory-related performance issues is again examined in detail. However, we also strive to maintain a high-level overview of the entire finite element solution procedure. Whereas the two stages of finite element computations are usually offloaded to a GPU separately, we consider how to couple these computations together to avoid memory traffic bottlenecks due to CPU-GPU data transfers.

Question 3. What are the most important memory-related influences on performance when offloading finite element solvers to a GPU?

After offloading the desired calculations to a GPU, it seems only natural to compare the resulting performance to that of a typical multi-core CPU system.

Question 4. How does a GPU compare to a typical multi-core CPU in terms of performance for a realistic finite element problem?

A convincing answer to this question ought to be based on optimised implementations for both platforms. Thus, part of our strategy is to first produce an optimised finite element assembly implementation for multi-core CPUs in Paper II. Thereafter, we consider GPU acceleration of linear system assembly in Paper III. In this way, a fair comparison can be made between CPU and GPU performance in the latter paper. Moreover, we aim to use automated code generation, so that we can consider a more complicated, real-world problem in the third paper.

The final research question stems from our goal to make high-performance GPU-based finite element solvers more accessible with the help of automated code generation.

Question 5. How can automated code generation be used to implement GPU-based finite element solvers for a range of PDEs?

We are not aware of any previous work that has successfully provided a complete solution that combines automated code generation with fully GPU-accelerated finite element solvers, including both GPU-based assembly and solution of linear systems.

However, the PDE solver framework FEniCS [Logg et al. 2012] has previously demonstrated its worth by using automated code generation for parallel finite element solvers for large-scale problems on clusters of multi-core CPUs. One of the novel contributions of this thesis is therefore our extension of FEniCS in Paper III, which enables GPU-based finite element assembly and solvers in a way that complements the framework's existing functionality.

1.3 Summary of research papers

This section summarises three research papers that form the backbone of this thesis.

1.3.1 Summary of Paper I

Trotter, J. D., J. Langguth and X. Cai (2020). "Cache simulation for irregular memory traffic on multi-core CPUs: Case study on performance models for sparse matrix-vector multiplication". In: *Journal of Parallel and Distributed Computing*, *144*, pp. 189–205. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2020.05.020.

The first research paper of this thesis concerns the parallel performance of computations that prominently feature irregular memory access patterns. In the context of finite element computations, irregular memory accesses naturally arise due to using unstructured meshes and sparse matrices. The paper focuses particularly on the latter by using sparse matrix-vector multiplication (SpMV) as the primary example of an irregular computational kernel.

Background and motivation. SpMV is one of the main components of iterative linear solvers [Saad 2003], which are frequently used to solve large-scale finite element problems. It also appears in other situations, such as graph algorithms [Davis 2019; Kepner and Gilbert 2011], or explicit schemes for time-dependent PDEs (e.g., Langguth et al. [2015a] and Langguth et al. [2015b]).

On the surface, SpMV is deceptively simple. The most common variants of this kernel can be expressed in only a few lines of code and the number of floating-point arithmetic operations to be carried out is simply twice the number of non-zero matrix values. However, the performance observed in practice may vary considerably as a result of irregular memory accesses.

In spite of its simplicity, there is a substantial body of literature covering optimisations for SpMV. For example, some apply techniques such as register and cache blocking [Nishtala et al. 2007; Pinar and Heath 1999; Vuduc et al. 2002] or compression [Willcock and Lumsdaine 2006] to standard sparse matrix formats such as CSR (see Section 1.1.1), whereas others propose alternative sparse matrix storage formats [Buluç et al. 2009; Haase et al. 2007; Kreutzer et al. 2014; W. Liu and Vinter 2015; X. Liu et al. 2013; Yzelman and Bisseling 2012].

In most cases, SpMV algorithms are evaluated experimentally by trying them out on a representative set of matrices, such as the SuiteSparse Matrix Collection [Davis and Hu 2011]. Examples include the informative overviews of Goumas et al. [2009] and Williams et al. [2009a] regarding SpMV on multi-core CPUs. Occasionally, performance is tied to best case estimates of memory traffic based on the number of rows, columns and non-zeros of a given matrix [Bender et al. 2010; Heras et al. 2001; Temam and Jalby 1992; Vuduc et al. 2002]. These results establish upper bounds on achievable performance. But the performance observed in practice is often much worse, especially for irregular matrices. Similarly, arithmetic intensity is easily deduced (about 0.1 flop/B for a standard, CSR-based SpMV kernel), yet a simple bottleneck analysis, like the roofline model [Williams et al. 2009b], frequently misses the mark.

Cache simulation. Existing memory traffic estimates for SpMV disregard the structure of a matrix, that is, the locations of its non-zeros. These estimates are simple to derive, depending only on the number of rows, columns and non-zeros. However, they can be highly inaccurate, especially for irregular matrices.

Therefore, inspired partly by analytical cache models [Agarwal et al. 1989; Frigo et al. 2012] and trace-driven memory simulation [Uhlig and Mudge 1997], Paper I proposes a trace-driven cache simulation technique for multi-core CPUs with multiple levels of private and shared caches. The method is presented in general terms and is applicable to any irregular, parallel computation, even though our main focus is on SpMV. Furthermore, the difference compared to previous analytical cache models and trace-driven approaches, see Burgess and Giles [1997], Heras et al. [2001], and Yzelman and Bisseling [2009], is the incorporation of multiple memory hierarchy levels as well as shared caches.

The method presented in the paper is aimed at a typical multi-core CPU's memory hierarchy, as outlined in Section 1.1.2. We assume that caches are fully associative and use a least recently used policy when evicting cache lines to make room for new ones.

Also, the method accounts for cache lines that are brought to a cache as a direct result of load or store instructions issued by a CPU without performing any prefetching.

Furthermore, each cache is considered to be inclusive of higher-level caches, which means that a given cache always contains cache lines that are also held in smaller caches, closer to the CPU. For example, anything that is found in a first-level cache will also be present in a second- and third-level caches. As a result, the cache simulation can be carried out independently for each cache.

In broad strokes, the cache simulation runs through a sequence of memory accesses performed by a given algorithm to estimate memory traffic volumes associated with each memory hierarchy level. Each cache is represented by a least recently used (LRU) list to keep track of cache lines that were accessed most recently. As the simulation progresses, it counts the number of cache misses that occur as new cache lines are placed at the front of the LRU list and the oldest entries are evicted. The final memory traffic volume is simply the cache line size (64 bytes, in most cases) multiplied by the number of cache misses.

Finally, consider a cache that is shared by several CPU cores, such as the third-level caches of most multi-core CPUs. In practice, the load and store requests to a shared cache will be received in some unpredictable order. Threads running on different cores typically proceed at different speeds for all sorts of reasons, such as thread scheduling or memory latencies. In any case, we assume that memory accesses from different CPU cores are interleaved. In other words, CPU cores are treated as though they issue load and store instructions in a round-robin fashion.

Numerical experiments. In the paper, three multi-core CPU system are used, including Intel and AMD CPUs. The main example in our numerical experiments is an OpenMP-parallel SpMV kernel using matrices in the CSR format, (see Section 1.1.1). Thus, a small subset of the SuiteSparse Matrix Collection [Davis and Hu 2011] was selected, representing various application domains, such as circuit simulation, combinatorial problems, computational fluid dynamics, and more. Some of these matrices are shown in Figure 1.4.

Estimates of memory traffic volumes for first, second- and third-level caches, as well as main memory, are produced by running the cache simulation for a standard OpenMP-parallel implementation of SpMV for matrices in CSR format. In addition, the SpMV algorithm is itself run to measure its execution time and performance.

For the Intel systems, hardware performance monitoring facilities are used to extract the actual volumes of memory traffic for each memory hierarchy level during SpMV. These hardware measurements are directly compared to estimates produced by the cache simulation.

To highlight some results, Table 1.1 shows estimated and measured memory traffic volumes on a dual-socket Intel Sandy Bridge system for the matrices in Figure 1.4. Overall, the estimates from the cache simulation are close to the memory traffic measurements. The error is less than 10 % even for very challenging and irregular matrices, such as “GL7d19”, “sx-stackoverflow” and “Lynx68”.

The cache simulation mostly underestimates the memory traffic volume by a small amount, for example, due to ignoring conflict misses. On the other hand, the memory

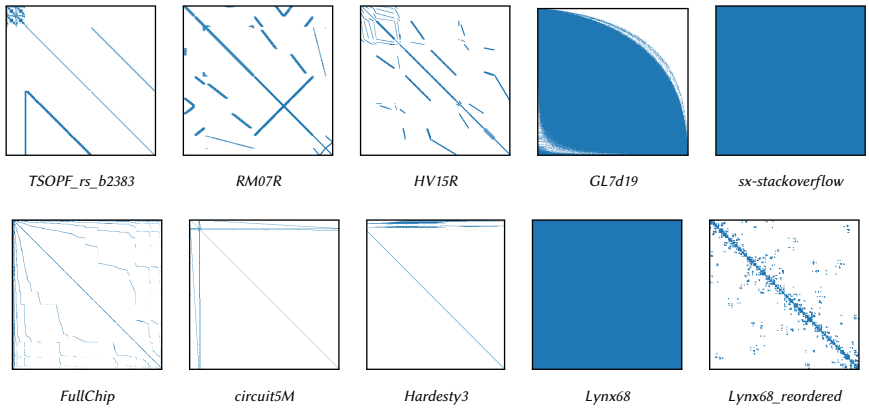


Figure 1.4. Matrix sparsity patterns. The matrices "GL7d19" and "Hardesty3" are rectangular, whereas the others are square.

Table 1.1. Memory traffic of OpenMP-parallel CSR SpMV between shared L3 cache and main memory (DRAM) using 16 threads on dual-socket Intel Xeon E5-2650 (Sandy Bridge) CPUs.

Matrix	Measured [MiB]	Estimated [MiB]	Error ([%])
TSOPF_RS_b2383	185	185	(+0.0 %)
RM07R	451	439	(−2.7 %)
HV15R	3413	3357	(−1.6 %)
GL7d19	617	563	(−8.8 %)
sx-stackoverflow	795	732	(−7.9 %)
FullChip	449	438	(−2.4 %)
circuit5M	895	917	(+2.5 %)
Hardesty3	620	618	(−0.3 %)
Lynx68	4402	4290	(−2.5 %)
Lynx68_reordered	1481	1463	(−1.2 %)

traffic is overestimated for "circuit5M", which is possibly a result of threads competing for the shared L3 cache. Recall that the cache simulation assumes that load and store requests are submitted by CPU cores in a simple round-robin scheme, but this is not really the case in practice.

The paper also contains further examples and detailed explanations about discrepancies between measured memory traffic volumes and estimates based on the cache simulation approach.

For the sake of comparison, a much simpler "pen-and-paper" estimate for the "Lynx68" matrix places the memory traffic between 1401 MiB (about 20 B per non-zero plus 20 B per row) and 8134 MiB (about 76 B per non-zero plus 12 B per row). This is far away from the actual memory traffic that is measured between L3 cache and main memory. Further results in the paper show that such simple best- and worst case

Table 1.2. Performance of OpenMP-parallel CSR SpMV using 48 threads on dual-socket Intel Xeon Platinum 8168 (Skylake) CPUs. This includes actual measured performance, best case performance estimates based on "pen-and-paper" data traffic estimates, and performance estimates based on memory traffic from cache simulation.

	Measured	Best case		Cache simulation	
Matrix	[Gflop/s]	[Gflop/s]	(Error [%])	[Gflop/s]	(Error [%])
TSOPF_RS_b2383	13.36	30.63	(+129 %)	16.64	(+25 %)
RM07R	31.81	30.24	(−5 %)	31.01	(−3 %)
HV15R	29.11	30.39	(+4 %)	31.96	(+10 %)
GL7d19	13.46	28.31	(+110 %)	12.72	(−6 %)
sx-stackoverflow	2.70	27.46	(+916 %)	2.24	(−17 %)
FullChip	4.71	25.91	(+450 %)	4.96	(+5 %)
circuit5M	3.87	30.28	(+683 %)	3.85	(−0 %)
Hardesty3	16.26	23.15	(+42 %)	24.68	(+52 %)
Lynx68	9.31	27.91	(+200 %)	11.48	(+23 %)
Lynx68_reordered	23.33	27.91	(+20 %)	25.68	(+10 %)

estimates are even more problematic for the smaller L1 and L2 caches.

Next, the estimated memory traffic volumes are used to predict SpMV performance, under the assumption that memory and cache bandwidth are the main performance bottlenecks. Thus, we first need to measure realistic memory and cache bandwidths for each of our multi-core CPUs. These measurements are obtained from a slightly modified STREAM benchmark [McCalpin 1995]. Dividing each estimated memory traffic volume by the corresponding bandwidth results in a lower bound on execution time (or upper bound on performance) for SpMV with a given matrix.

Table 1.2 shows measured and estimated performance for the same matrices as before. These results demonstrate how the cache simulation produces better performance predictions than simpler "pen-and-paper" estimates, especially for irregular matrices. More specifically, best case predictions miss by a factor of 5 to 10 times in cases such as "sx-stackoverflow", "FullChip" and "circuit5M". The cache simulation, on the other hand, are within 20 % of the actual performance. Generally speaking, our cache simulation estimates are always about the same quality or better than the best case estimates.

The best case estimates are almost always too optimistic, whereas the cache simulation sometimes underestimates performance. This may happen due to over- or underestimating memory traffic at different levels of a memory hierarchy, and typically requires closer investigation in each individual case.

Conclusions. For SpMV, performance depends heavily on the sparsity pattern of the matrix in question. The proposed cache simulation technique can be used to predict the performance of SpMV even in the presence of highly irregular sparsity patterns. More generally, the presented method is not specific to SpMV, but can be applied to other irregular, memory bandwidth-limited computations.

1.3.2 Summary of Paper II

Trotter, J. D., X. Cai and S. W. Funke. "On memory traffic and optimisations for low-order finite element assembly algorithms on multi-core CPUs".
Submitted for publication.

Shifting our attention away from sparse matrix-vector multiplication, the second paper of this thesis is about implementing finite element assembly algorithms on multi-core CPUs. However, we retain a strong emphasis on memory traffic, an aspect that is brought over from the first paper and continues to form a common thread throughout this thesis. The second paper also incorporates an important element of performance modelling, which is used to evaluate performance optimisations that are suggested.

Motivation. Most existing work on finite element assembly tends to focus on arithmetic operations, especially in connection with computing element vectors and matrices, as outlined in Section 1.1.1. Some techniques that have been proposed are sum factorisation [Bolis et al. 2014; Cantwell et al. 2011; Kronbichler and Kormann 2012; Vos et al. 2010], manual vectorisation [Sun et al. 2020], low-level loop optimisations [Homolya et al. 2018; Luporini et al. 2017; Luporini et al. 2015; Ølgaard and Wells 2010], or tensor representation and other means of simplifying expressions [Alnæs and Mardal 2010; Kirby and Logg 2006; Rognes et al. 2009; Russell and Kelly 2013].

If memory traffic is mentioned in connection with finite element computations, it is usually with respect to the well established practice of reordering sparse matrices. For instance, the Cuthill-McKee algorithm [Cuthill and McKee 1969] or nested dissection [George 1973; George and Mcintyre 1978] are often used. Initially, such techniques were motivated by reducing fill-in when using direct solvers. It has also been firmly established that reordering can speed up SpMV operations of iterative solvers [Oliker et al. 2002; Pichel et al. 2005; Pichel et al. 2012]. For example, the Cuthill-McKee ordering reduces matrix bandwidth, leading to improved cache reuse during SpMV. Reordering is often combined with other techniques for reducing memory traffic that have already been mentioned in connection with Paper I, such as register and cache blocking [Nishtala et al. 2007; Pinar and Heath 1999; Vuduc et al. 2002].

Unlike previous work on linear solvers, there appears to be little in the existing literature about the influence of memory traffic on finite element assembly. Our second paper improves the situation by conducting a thorough analysis and benchmarking of the standard, cellwise, global finite element assembly algorithm, while paying special attention to the memory traffic involved.

Background. Paper II begins with the commonly used, cellwise algorithm for global assembly of sparse matrices, which is outlined in Section 1.1.1. The paper breaks the algorithm down into the following four kernels:

- Gathering vertex coordinates
- Transforming to a reference cell
- Computing element matrices
- Scattering results to a global matrix

Of these kernels, the first and last one involve reading and writing data to and from memory, respectively. Ideally, the other two involve only floating-point arithmetic on data that is already located in registers or perhaps in a first-level cache.

The details of each kernel depend on the type of mesh cells and finite elements that are used. The calculation of element matrices in the third kernel depends especially on the underlying PDE that is targeted by the finite element solver. Examples studied in the paper revolve around Poisson's equation with first- and second-order Lagrange elements on tetrahedral meshes.

Memory traffic analysis. Having identified the sources of memory traffic arising during assembly, we proceed to quantify the induced memory traffic volumes. These results take the form of best- and worst-case estimates that are later used to judge the effectiveness of mesh reordering and other optimisations. A brief summary is presented below, while a more detailed derivation is found in the paper.

First, we show that gathering d coordinates of m vertices for each cell in a mesh with N cells results in $N \times \left(\lceil m/(2w) \rceil + m \lceil d/w \rceil \right)$ load instructions, where vectorisation is used to load w consecutive coordinate values or $2w$ integers. Also, for a cache where a cache line holds L coordinate values or $2L$ integers, the number of cache misses is at least $\lceil Nm/(2L) \rceil + \lceil Md/L \rceil$ and at most $\lceil Nm/(2L) \rceil + Nm \lceil d/L + 1 \rceil$, where M is the number of vertices in the mesh.

As a practical example, consider the largest of the meshes used in the paper, with $M = 3.02 \times 10^6$ vertices and $N = 16.91 \times 10^6$ cells. Since each cell is a tetrahedron, we have $m = 4$ vertices per cell and $d = 3$ coordinates per vertex. Assuming a cache line size of $L = 8$ (64 bytes), the ratio between the worst- and best case memory traffic is about 26. Moreover, the paper's numerical experiments demonstrate that memory traffic is very far from the best case, if one uses a poor mesh ordering. Unfortunately, such low quality mesh orderings are the default for some mesh generators.

A similar analysis is carried out for two different versions of the last kernel, which is responsible for scattering element matrices to a global matrix. The first variant performs a binary search for each element matrix entry to find the location in the array of global matrix values that needs to be updated. The second version, on the other hand, uses a precomputed lookup table to avoid carrying out binary searches during assembly. As a result of this optimisation, fewer loads are issued overall, though some additional storage and memory traffic is needed for the lookup table itself.

Mesh reordering. Numerical experiments presented in the paper show that a poor ordering of the mesh or global degrees of freedom leads to one or two orders of magnitude more memory traffic than the best case described above. However, it is also shown that ordering the mesh vertices according to the reverse Cuthill-McKee algorithm [Cuthill and McKee 1969], and at the same time placing the mesh cells in lexicographic order, leads to memory traffic volumes close to the best case.

More specifically, when gathering the vertex coordinates of one particular large, tetrahedral mesh ("Cardiac mesh 20"), the memory traffic is reduced from 4202 to 378 MB after reordering. In comparison, the best case estimate is 342 MB. The performance increases accordingly from about 22 million to 260 million cells per second

(Mcell/s). Reordering has a similar effect on the scattering of element matrices to a global matrix.

Lookup table. Beyond estimating memory traffic volumes and experimentally assessing the impact of mesh reordering, the paper also evaluates the use of a lookup table for the scattering kernel, as well as a slightly modified algorithm that assembles a matrix row by row.

Switching from binary searches to using a lookup table leads to a speedup of about 24 times. This is a huge performance improvement, which cannot be explained solely in terms of the observed memory traffic volume. Instead, there seems to be a performance bottleneck related to memory latency that disappears whenever the search procedure is eliminated. This is highly plausible, if we keep in mind that a binary search entails some amount of branching logic, which is typically prone to memory latency issues.

Rowwise assembly. Rowwise assembly first becomes significant as a means to perform assembly in parallel using shared memory without the need for synchronisation between threads. In contrast, a cellwise assembly must prevent data races from different threads simultaneously updating the same global matrix values. This is usually achieved by atomic operations, which can be very expensive.

The paper compares the parallel performance and scalability of cellwise and rowwise algorithms for assembling matrices with respect to two basic variational forms using first- and second-order Lagrange elements. Moreover, three different multi-core CPUs are considered. In short, rowwise assembly performs best and scales almost perfectly, since it avoids atomic operations and displays good cache reuse.

Ultimately, a parallel, rowwise assembly of a matrix for Poisson's equation on a large, unstructured, tetrahedral mesh, reaches about 390 Mcell/s using 64 cores on a dual-socket AMD Epyc CPU.

Conclusions. Memory traffic is a deciding factor during assembly of linear systems for low-order finite element methods. Similar to SpMV, there are a great deal of irregular memory accesses. Optimisation strategies should be selected accordingly. For instance, memory latency bottlenecks are mitigated by replacing binary searches with a precomputed lookup table. Also, performing parallel assembly rowwise has the benefit of improved cache reuse as well as avoiding costly thread synchronisation.

1.3.3 Summary of Paper III

Trotter, J. D., J. Langguth and X. Cai. "Leveraging GPU-accelerated finite element computation with automated code generation: A holistic approach".
Submitted for publication.

In Paper III, we offload finite element solvers to a GPU. This is achieved by extending the FEniCS PDE solver framework with automatic generation of GPU code. In this way, we supplement existing functionality and advanced features of FEniCS with auto-generated, GPU-based finite element solvers.

The implementation is informed by the knowledge gained in Paper II. Moreover, optimisations are carried out to seamlessly integrate the assembly and solution stages, so that unneeded CPU-GPU data transfers are avoided.

Finally, two different test cases are used to validate the correctness and performance of our implementation.

Background. The open-source PDE solver frameworks FEniCS [Logg et al. 2012] and Firedrake [Rathgeber et al. 2016] use automated code generation to deliver high performance, while at the same time remaining accessible to domain scientists. This is achieved through using the Unified Form Language (UFL) [Alnæs et al. 2014], as described briefly in Section 1.1.4.

FEniCS and Firedrake are currently designed for parallel, CPU-based, distributed-memory computations. Meanwhile, researchers have shown that GPUs can be a suitable fit for solving linear systems [Anzt et al. 2020; Anzt et al. 2017; Dongarra et al. 2014; Naumov et al. 2015] and accelerating other aspects of finite element methods, including calculation of element matrices [Banaś et al. 2014; Knepley and Terrel 2013] and assembly of global matrices [Cecka et al. 2010; Reguly and Giles 2015]. Entire finite element solvers have also been offloaded to GPUs [Fu et al. 2014; Pichler and Haase 2017]. Some GPU-accelerated finite element solvers are based on matrix-free and related methods [Kronbichler and Ljungkvist 2019; Ljungkvist 2014; Markall et al. 2013].

Automated code generation for GPUs. Paper III starts with a short overview of using FEniCS to solve PDEs. As described in Section 1.1.1, implementations of problem-specific computational kernels are needed to compute element vectors and matrices. Paper II studied handwritten examples of such kernels as a part of analysing common finite element assembly algorithms.

Alternatively, these kernels can be automatically generated by FEniCS from Unified Form Language (UFL) code, for example, as shown in Algorithm 1.1 in Section 1.1.1. The FEniCS form compiler (FFC) translates UFL code into C code for computing element vectors and matrices. This is done automatically if the UFL code is embedded in a Python application. Otherwise, the user invokes the form compiler to generate code, which can later be used from a C++ application. Further details on the form compiler are given in the FEniCS book [Logg et al. 2012].

Our effort to offload assembly to a GPU is based on NVIDIA's CUDA framework for GPU computing [NVIDIA Corporation 2020b], which was briefly introduced in Section 1.1.3. The first contribution of the paper is to extend FFC to emit code that can be compiled by the CUDA C++ compiler. Technically speaking, we modify FFC to get a string of CUDA C++ code, which is later fed to NVIDIA's runtime compilation API (NVRTC) [NVIDIA Corporation 2019]. The end result is that auto-generated functions for computing element vectors and matrices can be launched as CUDA kernels to run on a GPU, or called from other CUDA kernels running on a GPU.

GPU-based assembly. The current CPU-based assembly algorithm in FEniCS first partitions the mesh to distribute it among CPUs. A global matrix and vector are also

distributed based on the partitioning. Each CPU assembles part of the global matrix and vector corresponding to its own portion of the mesh.

Assembly is carried out cell by cell, much in the same way as described in Paper II. In short, after reading a cell's vertex coordinates—and maybe some problem-dependent constants or coefficient values—the appropriate auto-generated kernel is invoked to compute an element vector or matrix. The final step is to add the newly computed values to a global vector or matrix. In the case of a matrix, this part is done by PETSc [Balay et al. 2019], a library that FEniCS uses for its linear solvers and for working with sparse matrices.

Based on the current finite element assembly procedure in FEniCS, the paper starts with the easiest way of offloading assembly to a GPU. Thereafter, several improvements are considered. More specifically, the calculation of element vectors and matrices is moved to a GPU first, but the CPU is still used to add each element vector or matrix to its global counterpart. The fact that PETSc is used in the same way as before makes this strategy easy to implement in FEniCS.

Unfortunately, the above approach results in virtually no acceleration. One of the main reasons is that an excessive amount of data is moved between CPU and GPU memory. A far better proposition is therefore to offload the entire global assembly routine, including the scattering of element matrices to a global matrix. Detailed examples of auto-generated CUDA C++ kernels for both of these methods are given in the paper.

Once a GPU-based global assembly is in place, the paper discusses three code optimisations. Two of these were also considered in Paper II in connection with finite element assembly on multi-core CPUs, namely using a lookup table to scatter element matrix values and assembling a global matrix row by row.

The motivation behind these optimisations is the same as in the previous paper, only further strengthened by findings that come from investigating the auto-generated CUDA kernels with NVIDIA's profiling tools. In particular, profiling indicates that memory latency associated with binary searches is a significant performance bottleneck. Using a lookup table to scatter element matrices alleviates the problem.

Rowwise assembly provides some of the same benefits for a GPU as it did for multi-core CPUs in Paper II.

The third optimisation applies when offloading both assembly and linear solver to a GPU. In this case, we make some changes to PETSc to avoid certain unfortunate CPU-GPU data transfers. Numerical experiments presented in the paper show that these transfers slow down the GPU-based assembly by about 3 to 5 times. In other words, it is not enough to separately offload assembly and solution of linear systems and hope for the best. Special care is taken to integrate these two steps to provide seamless GPU acceleration.

Numerical experiments. Several experiments are carried out to assess the performance of assembling sparse matrices for Poisson's equation on large, unstructured, tetrahedral meshes. First, FEniCS's CPU-based assembly on a dual-socket AMD Epyc 7601 CPU assembles about 58 million cells per second (Mcell/s). In comparison, GPU-

based global assembly with an NVIDIA V100 GPU yields a performance of about 450 to 1250 Mcell/s.

If a lookup table is used in the scattering kernel of the GPU-based global assembly, then performance improves by about 70 to 90 %. Rowwise assembly results in another 70 % improvement on top of that. In the end, the NVIDIA V100 assembles 1600 Mcell/s.

Comparing to the results of Paper II, where we recall that AMD Epyc attained 390 Mcell/s, the final outcome is a speedup of about 4 for the auto-generated GPU-based assembly over a highly optimised multi-core CPU version.

Hyperelasticity. In addition to Poisson's equation, the paper draws on a more advanced example in solid mechanics from Ølgaard and Wells [2012], featuring a non-linear, vector PDE with spatially varying coefficients. The description provided in the paper is very concise due to limited space, so we supplement it with a few more details here.

In short, the problem is posed as finding a displacement field $u\colon \Omega \to \mathbf{R}^3$, for a solid body occupying a polygonal domain $\Omega \subset \mathbf{R}^3$. This is done by minimising the total potential energy,

$$\Pi(u) = \int_\Omega \psi(u)\,\mathrm{d}x - \int_\Omega B \cdot u\,\mathrm{d}x - \int_{\partial\Omega} T \cdot u\,\mathrm{d}s, \tag{1.5}$$

where B is a body force and T is a traction force on the boundary. Also, ψ is the stored strain energy density function, which, in this case, is based on a Neo-Hookean elastic stored energy model,

$$\psi = \frac{\mu}{2}(I_c - 3) - \mu\ln(J) + \frac{\lambda}{2}\ln(J)^2. \tag{1.6}$$

Here μ and λ are Lamé parameters that depend on the material, $J = \det(F)$, $I_c = \mathrm{trace}(C)$, $C = F^{\mathsf{T}}F$ is the right Cauchy-Green tensor, and $F = I + \nabla u$ is the deformation gradient.

In our example, the Lamé parameters are set to $\mu = \frac{E}{2(1+\nu)}$, $\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}$, where $E = 10$ and $\nu = 0.3$. Also, a constant body force of $(0, -0.1, 0)$ and constant traction force of $(0.01, 0, 0)$ were used.

The variational formulation of the above minimisation problem is obtained from directional derivatives of the energy functional Π, leading to the following pair of forms,

$$L(u; v) = D_v\Pi = \lim_{\epsilon\to 0} \frac{d\Pi(u + \epsilon v)}{d\epsilon} \tag{1.7}$$

and

$$a(u; \delta u, v) = D_{\delta u}L = \lim_{\epsilon\to 0} \frac{dL(u + \epsilon\delta u; v)}{d\epsilon}. \tag{1.8}$$

The UFL code for the hyperelasticity model is shown in Algorithm 1.2.

A Newton solver is used for the non-linear system of equations. Also, the linear system that arises during each Newton iteration is solved using the conjugate gradient method without preconditioning.

```
# Function spaces and functions
e = VectorElement("Lagrange", tetrahedron, 1)
du = TrialFunction(e) # Incremental displacement
v = TestFunction(e)   # Test function
u = Coefficient(e)    # Displacement
B = Coefficient(e)    # Body force per unit volume
T = Coefficient(e)    # Boundary traction force

# Kinematics
d = len(u)            # Mesh dimension (d=3)
I = Identity(d)       # Identity tensor
F = I + grad(u)       # Deformation gradient
C = F.T*F             # Right Cauchy-Green tensor

# Invariants of deformation tensors
Ic = tr(C)
J  = det(F)

# Elasticity parameters
mu    = Constant(tetrahedron)
lmbda = Constant(tetrahedron)

# Stored strain energy density for
# a compressible neo-Hookean model
psi = ((mu/2)*(Ic - 3) - mu*ln(J) +
       (lmbda/2)*(ln(J))**2)

# Total potential energy
Pi = psi*dx - inner(B, u)*dx - inner(T, u)*ds

# The first variation and its Jacobian are
# automatically computed with `derivative`
F = derivative(Pi, u, v)
J = derivative(F, u, du)
```

Algorithm 1.2. UFL code for a hyperelasticity model.

Briefly, assembly is about five times faster and the linear solver is twice as fast on NVIDIA V100 compared to AMD Epyc. There is an additional cost associated with GPU offloading due to certain initial data transfers from CPU to GPU and computing the lookup table used to scatter element matrices. Taking this cost into account lessens the speedup of the assembly portion somewhat, though GPU-accelerated assembly is still faster overall. Note that most CPU-GPU data transfers are performed only once, and are therefore expected to become negligible in the case of a time-dependent problem.

Conclusions. The FEniCS form compiler is adapted to also generate CUDA C++ code for fundamental, problem-specific kernels that are used to assemble linear systems of equations for finite element methods. These kernels form the basis for offloading global finite element assembly to GPUs.

Whenever these calculations are offloaded, it becomes critical to avoid unnecessarily transferring data between CPU and GPU. The penalty associated with such transfers can easily overcome performance gains obtained by using a GPU to carry out calculations.

The performance of GPU-based assembly benefits from using a lookup table to scatter element matrices to a global matrix and performing assembly rowwise, the same as for multi-core CPUs in Paper II. GPU-based assembly is demonstrated for an advanced, non-linear, vector PDE with variable coefficients. Combined with a GPU-based linear solver, the final result is a substantial speedup over a typical dual-socket multi-core CPU.

1.4 Discussion and conclusions

Returning once more to the research questions in Section 1.2, we discuss to what extent they are answered by the research summarised in the previous section. In addition to conclusions surrounding each research question, we also mention some opportunities for future work.

Finite element assembly. Recall that our first question concerns the performance of finite element assembly in relation to the use of memory during such calculations. This aspect of finite element computations is covered in Papers II and III. Both papers present memory-related optimisations that are motivated by in-depth analysis and profiling. Moreover, the suggested optimisations have a tremendous performance impact on the canonical finite element problem of solving Poisson's equation on unstructured, tetrahedral meshes.

The following guidelines summarise our findings regarding this point:

- A precomputed lookup table should be used to mitigate memory latency issues associated with binary searches when scattering local element matrices to a global matrix.
- Rowwise assembly should be considered to improve cache reuse of global matrix values and to avoid expensive atomic operations that are otherwise needed to prevent race conditions.

Paper II also reaffirms the significance of well known practices surrounding mesh reordering. With these conclusions, we consider the first research question to be answered in a satisfactory way.

With the exception of the non-linear hyperelasticity problem in Paper III, most of the work presented in this thesis is centred on Poisson's equation as an example PDE problem. But even if another PDE is considered, possibly demanding more floating-point arithmetic per mesh cell, then it is still reasonable to believe that the above-mentioned optimisations will remain helpful. Though the balance between memory traffic and floating-point operations may shift, it is still important to prevent memory latencies from stalling progress and causing floating-point units to stand idle.

Also, if element matrices become larger, due to solving vector PDEs or increasing the order of finite elements, then the scattering of element matrices to a global matrix triggers much more memory traffic than before. On the other hand, as the element order increases, so does the amount of floating-point computation. At that point, it may anyway be a good idea to resort to strategies such as sum factorisation or

matrix-free methods [Kronbichler and Kormann 2012; Sun et al. 2020], which may improve floating-point performance.

Sparse matrix-vector multiplication. Our second research question is related to understanding the performance of SpMV on multi-core CPUs, which is the topic of Paper I. The paper goes further by developing a method that is applicable also to other irregular, memory bandwidth-limited computational kernels. However, the main effort is to quantitatively model SpMV performance, which goes towards understanding the performance of iterative solvers for sparse linear systems.

The quantitative performance model presented in Paper I is able to assess the cost of an SpMV operation even for highly irregular matrices. These are cases where simple best case memory traffic estimates are far too optimistic. Our method therefore represents one possible approach to investigate SpMV performance, which stresses the importance of accounting for matrix structure.

The fact that the actual hardware itself is not needed to produce the desired memory traffic estimates might be considered another advantage of this method. This aspect is not discussed in the work presented here, but has been useful in some related, unpublished work, where we consider the so-called format selection problem. The idea is to determine which of several eligible sparse matrix storage formats that results in the best performance for a given matrix on a particular hardware with respect to some sparse matrix operation, such as SpMV. In this case, we were able to use the cache simulation approach for an ARM-based multi-core CPU system to gain a better understanding of which sparse matrix format to choose, even before the target ARM system became available to us.

For a given matrix, the cache simulation method can be used to point to a particular memory hierarchy level that constitutes a performance bottleneck. While this may not immediately suggest good optimisations or matrix reordering strategies, it is conceivable that a matrix reordering strategy, or at least some useful heuristics, could be derived from the simulation approach.

The performance estimates presented in Paper I are based on full knowledge of the sparsity pattern of a matrix. Although this enables accurate predictions, a disadvantage is that the entire matrix must be processed. The cache simulation can thus be somewhat expensive to compute. A possible approach is to use only a part of the matrix, for example, by sampling the sparsity pattern, to strike a compromise between accuracy and the time and effort needed to obtain a prediction.

Finally, Paper I also explores some of the known limitations of the proposed cache simulation method. In particular, an alternative SpMV kernel based on matrices in the so-called coordinate (COO) format proves to be challenging because of irregular writes to memory. This scenario is not properly dealt with by the current cache simulation approach, since it does not account for additional memory traffic due to cache coherency effects like false sharing of cache lines. However, there is nothing that, in principle, prevents a more advanced cache simulation from incorporating more features, such as cache coherency or associativity. This is one possible avenue for future work.

GPU-based assembly. Turning to memory-related performance considerations of GPU-based assembly, it turns out that the same techniques that were considered for multi-core CPUs in Paper II are also applicable to GPUs. More specifically, Paper III concludes that a precomputed lookup table should be used to scatter element matrices to a global matrix, even for GPU-based assembly. Furthermore, rowwise assembly can again improve performance. In this case, the primary benefit appears to be improved cache reuse.

More importantly, Paper III demonstrates the significance of avoiding CPU-GPU data transfers when offloading a finite element solver to a GPU. In this respect, we benefit from a complete overview of the entire finite element computation. This allows us to apply application-level optimisations that are not usually considered when assembly and solution of linear systems are separately offloaded to a GPU.

In the end, NVIDIA's performance profiling tools reveal that the only CPU-GPU data transfers remaining are a few values related to testing the convergence of the linear solver. These could conceivably be eliminated too, but they are so minuscule that we do not expect there to be any substantial performance benefit.

Another question raised in Section 1.2 is about comparing a GPU to a typical multi-core CPU. An answer to this question is likely to vary from case to case, depending on the precise details of the finite element problem that is studied. From this perspective, the question in general remains open and deserves further attention. However, as a result of the work in Paper III, we now have at our disposal the ability to automatically generate finite element kernels for GPUs. Naturally, these come in addition to the usual ones for multi-core CPUs. We are therefore in a much better position to carry out practical, experimental comparisons of CPU and GPU performance for a broad range of relevant and interesting problems.

Most of our attention has been devoted to finite element assembly, and the only linear solver we have considered so far is an unpreconditioned conjugate gradient method. Many finite element problems will benefit from more advanced linear solvers and preconditioning techniques. These are, in some cases, compulsory if an acceptable solution is to be found at all. The GPU acceleration of various preconditioning schemes and advanced linear solvers is currently a rapidly evolving field [Aliaga et al. 2019; Anzt et al. 2020; Anzt et al. 2017; Chen et al. 2018; Geveler et al. 2013; Li and Saad 2013; H. Liu et al. 2015; Naumov et al. 2015]. There are certainly interesting opportunities for future work in employing these kinds of linear solvers for finite element computations.

The concrete examples that are studied in Papers II and III of this thesis include Poisson's equation and a non-linear hyperelasticity problem. In these cases, an NVIDIA V100 GPU achieves a notable speedup over the best multi-core CPU results with respect to the overall finite element solver. The time used to assemble linear systems on the GPU is comparable to or less than the time spent by multi-core CPUs, even if GPU-specific initialisation and CPU-GPU data transfers are included. For the unpreconditioned conjugate gradient solver, where most of the time is spent, NVIDIA V100 demonstrates a clear speedup over dual-socket AMD Epyc and Intel Xeon CPUs.

We conclude tentatively that a high-powered GPU, such as NVIDIA V100, has a lot of potential to accelerate finite element solvers. But further research is needed to discern the effectiveness of using GPUs for finite element computations, particularly when more advanced linear solvers and preconditioning techniques are involved.

Automated code generation. The final research question concerns GPU-based finite element solvers and automated code generation. This matter is resolved in Paper III, where we provide a fully working implementation for GPU-based finite element computations by building on the automated code generation facilities of the FEniCS PDE solver framework.

In our view, FEniCS presents an ideal starting point for our implementation efforts. Not only has it been perhaps the foremost proving ground for combining automated code generation and finite element computations, but it has also become a popular framework with many users, collectively solving a huge variety of PDE problems. Moreover, FEniCS is already equipped to solve large-scale problems on clusters of multi-core CPUs.

In short, we have the great advantage of not having to start from scratch. Instead, we can employ a proven technology like FEniCS, where a lot of work has already been poured into optimising automatically generated finite element kernels.

In spite of its complexity, we have found a relatively straightforward way of implementing GPU acceleration that is orthogonal to the many other concerns that FEniCS deals with. We demonstrate in Paper III that our final implementation supports advanced use cases, such as non-linear, vector PDEs and any form of boundary conditions that can already be used in FEniCS.

Once more, we point out that successful GPU-acceleration depends on having a clear view of the entire finite element solver. With some work, we are able to carry out critical, application-level optimisations to eliminate unnecessary CPU-GPU data transfers and seamlessly couple the assembly and solution stages of a finite element solver.

There is potentially room for further improvements to the auto-generated GPU code beyond the substantial performance gains already demonstrated in Paper III. Also, there is a wide range of PDEs and applications to explore, possibly generating new ideas and opportunities for performance tuning and optimisations with respect to the automatically generated finite element kernels.

For the sake of convenience, Paper III only uses FEniCS through C++ applications that were written to benchmark the relevant PDE solvers. However, the more common approach is to employ FEniCS via its Python API. Further work may be required to ensure that the newly developed GPU acceleration features work equally well from Python. There is potentially a lot of value in doing so, since GPU-accelerated solvers will then become more easily available to most users.

Our current work investigates GPU-accelerated finite element solvers using a single GPU. Another aspect to consider in the future is therefore to combine GPU acceleration with FEniCS's MPI-based distributed memory computations. These features are complementary and ought to work well together, but further implementation work and performance optimisations may be required. This would represent a significant milestone in enabling automatically generated solvers for large-scale finite element problems.

Bibliography

Agarwal, A., M. Horowitz, and J. Hennessy (May 1989). "An Analytical Cache Model". In: *ACM Transactions on Computer Systems* 7.2, pp. 184–215. ISSN: 0734-2071. DOI: 10.1145/63404.63407.

Aliaga, J. I., E. Dufrechou, P. Ezzatti, and E. S. Quintana-Ortí (2019). "An efficient GPU version of the preconditioned GMRES method". In: *The Journal of Supercomputing* 75, pp. 1455–1469. ISSN: 0920-8542. DOI: 10.1007/s11227-018-2658-1.

Alnæs, M. S., A. Logg, K. B. Ølgaard, M. B. Rognes, and G. N. Wells (Feb. 2014). "Unified form language: A domain-specific language for weak formulations of partial differential equations". In: *ACM Trans. Math. Softw.* 40.2. ISSN: 0098-3500. DOI: 10.1145/2566630.

Alnæs, M. S. and K.-A. Mardal (Jan. 2010). "On the Efficiency of Symbolic Computations Combined with Code Generation for Finite Element Methods". In: *ACM Trans. Math. Softw.* 37.1. ISSN: 0098-3500. DOI: 10.1145/1644001.1644007.

Aneurisk-Team (June 2012). *AneuriskWeb project website*. Emory University, Department of Math&CS. URL: http://ecm2.mathcs.emory.edu/aneuriskweb.

Anzt, H., E. Boman, R. Falgout, P. Ghysels, M. Heroux, X. Li, L. Curfman McInnes, R. Tran Mills, S. Rajamanickam, K. Rupp, B. Smith, I. Yamazaki, and U. Meier Yang (2020). "Preparing sparse solvers for exascale computing". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378.2166. DOI: 10.1098/rsta.2019.0053.

Anzt, H., M. Gates, J. Dongarra, M. Kreutzer, G. Wellein, and M. Köhler (2017). "Preconditioned Krylov solvers on GPUs". In: *Parallel Computing* 68, pp. 32–44. ISSN: 0167-8191. DOI: 10.1016/j.parco.2017.05.006.

Arnold, D. N. and A. Logg (Nov. 2014). "Periodic Table of the Finite Elements". In: *SIAM News* 47 (9). URL: http://www.femtable.org/.

Axelsson, O. and V. A. Barker (2001). *Finite Element Solution of Boundary Value Problems: Theory and Computation*. Vol. 35. Classics in Applied Mathematics. SIAM. ISBN: 978-0-89871-499-9.

Balay, S., S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. Gropp, D. Karpeyev, D. Kaushik, M. Knepley, D. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. Smith, S. Zampini, H. Zhang, and H. Zhang (2019). *PETSc Web page*. URL: https://www.mcs.anl.gov/petsc.

Banaś, K., P. Płaszewski, and P. Macioł (2014). "Numerical integration on GPUs for higher order finite elements". In: *Computers & Mathematics with Applications* 67.6, pp. 1319–1344. ISSN: 0898-1221. DOI: 10.1016/j.camwa.2014.01.021.

Bender, M., G. Brodal, R. Fagerberg, R. Jacob, and E. Vicari (2010). "Optimal Sparse Matrix Dense Vector Multiplication in the I/O-Model". In: *Theory of Computing Systems* 47.4, pp. 934–962. ISSN: 1432-4350.

Bolis, A., C. D. Cantwell, R. M. Kirby, and S. J. Sherwin (Apr. 2014). "From h to p efficiently: optimal implementation strategies for explicit time-dependent problems

using the spectral/*hp* element method". In: *International Journal for Numerical Methods in Fluids* 75.8, pp. 591–607. ISSN: 0271-2091. DOI: 10.1002/fld.3909.

Buluç, A., J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson (2009). "Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks". In: *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*. SPAA '09. Calgary, AB, Canada: ACM, pp. 233–244. ISBN: 978-1-60558-606-9. DOI: 10.1145/1583991.1584053.

Burgess, D. A. and M. B. Giles (1997). "Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines". In: *Advances in Engineering Software* 28.3, pp. 189–201. ISSN: 0965-9978. DOI: 10.1016/S0965-9978(96)00039-7.

Cantwell, C. D., S. J. Sherwin, R. M. Kirby, and P. H. J. Kelly (Apr. 2011). "From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements". In: *Computers & Fluids* 43.1, pp. 23–28. ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2010.08.012.

Cecka, C., A. J. Lew, and E. Darve (Aug. 2010). "Assembly of finite element methods on graphics processors". In: *International Journal for Numerical Methods in Engineering* 85.5, pp. 640–669. DOI: 10.1002/nme.2989.

Che, S., M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron (Oct. 2008). "A performance study of general-purpose applications on graphics processors using CUDA". In: *J. Parallel Distrib. Comput.* 68.10, pp. 1370–1380.

Chen, Y., X. Tian, H. Liu, Z. Chen, B. Yang, W. Liao, P. Zhang, R. He, and M. Yang (2018). "Parallel ILU preconditioners in GPU computation". In: *Soft Computing* 22, pp. 8187–8205. ISSN: 1432-7643. DOI: 10.1007/s00500-017-2764-7.

Christen, M., O. Schenk, and H. Burkhart (2011). "PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures". In: *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*. IPDPS '11. USA: IEEE Computer Society, pp. 676–687. ISBN: 9780769543857. DOI: 10.1109/IPDPS.2011.70.

Ciarlet, P. G. (2002). *The Finite Element Method for Elliptic Problems*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics. ISBN: 0-89871-514-8.

Clint Whaley, R., A. Petitet, and J. J. Dongarra (2001). "Automated empirical optimizations of software and the ATLAS project". In: *Parallel Computing* 27.1, pp. 3–35. ISSN: 0167-8191. DOI: 10.1016/S0167-8191(00)00087-9.

Cuthill, E. and J. McKee (1969). "Reducing the Bandwidth of Sparse Symmetric Matrices". In: *Proceedings of the 1969 24th National Conference*. ACM '69. New York, NY, USA: Association for Computing Machinery, pp. 157–172. ISBN: 9781450374934. DOI: 10.1145/800195.805928.

Datta, K., M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick (2008). "Stencil Computation Optimization and Auto-Tuning on State-of-the-Art Multicore Architectures". In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC '08. Austin, Texas: IEEE Press. ISBN: 9781424428359. DOI: 10.5555/1413370.1413375.

Davis, T. (Dec. 2019). "Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra". In: *ACM Trans. Math. Softw.* 45.4. ISSN: 0098-3500. DOI: 10.1145/3322125.

Davis, T. and Y. Hu (2011). "The University of Florida Sparse Matrix Collection". In: *ACM Transactions on Mathematical Software (TOMS)* 38.1, pp. 1–25. ISSN: 1557-7295.

Dongarra, J., M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki (2014). "Accelerating Numerical Dense Linear Algebra Calculations with GPUs". In: *Numerical Computations with GPUs*, pp. 1–26.

Ern, A. and J.-L. Guermond (2004). *Theory and Practice of Finite Elements*. Applied Mathematical Sciences. Springer. ISBN: 0-387-20574-8.

Frigo, M., C. E. Leiserson, H. Prokop, and S. Ramachandran (Jan. 2012). "Cache-Oblivious Algorithms". In: *ACM Transactions on Algorithms* 8.1, 4:1–4:22. ISSN: 1549-6325. DOI: 10.1145/2071379.2071383.

Fu, Z., T. J. Lewis, R. M. Kirby, and R. T. Whitaker (Feb. 2014). "Architecting the finite element method pipeline for the GPU". In: *Journal of Computational and Applied Mathematics* 257, pp. 195–211. DOI: 10.1016/j.cam.2013.09.001.

George, A. (1973). "Nested Dissection of a Regular Finite Element Mesh". In: *SIAM Journal on Numerical Analysis* 10.2, pp. 345–363. ISSN: 0036-1429. DOI: 10.1137/0710032.

George, A. and D. R. Mcintyre (1978). "On the Application of the Minimum Degree Algorithm to Finite Element Systems". In: *SIAM Journal on Numerical Analysis* 15.1, pp. 90–112. ISSN: 0036-1429. DOI: 10.1007/BFb0064460.

Geveler, M., D. Ribbrock, D. Göddeke, P. Zajac, and S. Turek (2013). "Towards a complete FEM-based simulation toolkit on GPUs: Unstructured grid finite element geometric multigrid solvers with strong smoothers based on sparse approximate inverses". In: *Computers & Fluids* 80, pp. 327–332. ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2012.01.025.

Goumas, G., K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris (2009). "Performance evaluation of the sparse matrix-vector multiplication on modern architectures". In: *The Journal of Supercomputing* 50.1, pp. 36–77. ISSN: 0920-8542.

Guo, X. (Feb. 2019). *Best Practice Guide - AMD EPYC*. Ed. by O. W. Saastad. Partnership for Advanced Computing in Europe (PRACE). URL: https://prace-ri.eu/wp-content/uploads/Best-Practice-Guide_AMD.pdf.

Guo, X., C. Morales, O. W. Saastad, and A. Shamakina (Feb. 2019). *Best Practice Guide - ARM64*. Ed. by W. Rijks and V. Weinberg. Partnership for Advanced Computing in Europe (PRACE). URL: https://prace-ri.eu/wp-content/uploads/Best-Practice-Guide_ARM64.pdf.

Haase, G., M. Liebmann, and G. Plank (2007). "A Hilbert-order multiplication scheme for unstructured sparse matrices". In: *International Journal of Parallel, Emergent and Distributed Systems* 22.4, pp. 213–220. DOI: 10.1080/17445760601122084.

Han, D., S. Xu, L. Chen, and L. Huang (2011). "PADS: A Pattern-Driven Stencil Compiler-Based Tool for Reuse of Optimizations on GPGPUs". In: *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems*. ICPADS '11. USA: IEEE Computer Society, pp. 308–315. ISBN: 9780769545769. DOI: 10.1109/ICPADS.2011.94.

Heras, D., V. Blanco, J. Cabaleiro, and F. Rivera (2001). "Modeling and improving locality for the sparse-matrix-vector product on cache memories". In: *Future Generation Computer Systems* 18.1, pp. 55–67. ISSN: 0167-739X.

Holewinski, J., L.-N. Pouchet, and P. Sadayappan (2012). "High-Performance Code Generation for Stencil Computations on GPU Architectures". In: *Proceedings of the 26th ACM International Conference on Supercomputing*. ICS '12. San Servolo Island, Venice, Italy: Association for Computing Machinery, pp. 311–320. ISBN: 9781450313162. DOI: 10.1145/2304576.2304619.

Homolya, M., L. Mitchell, F. Luporini, and D. A. Ham (June 2018). "TSFC: A Structure-Preserving Form Compiler". In: *SIAM Journal on Scientific Computing* 40.3, pp. 401–428. ISSN: 1064-8275. DOI: 10.1137/17M1130642.

Intel Corporation (Apr. 2018). *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 248966-040. Intel Corporation.

Kepner, J. and J. Gilbert (2011). *Graph Algorithms in the Language of Linear Algebra*. Ed. by J. Kepner and J. Gilbert. Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9780898719918.

Khronos OpenCL Working Group (Apr. 2020). *The OpenCL Specification*. The Khronos Group. URL: https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf.

Kirby, R. C. and A. Logg (Sept. 2006). "A compiler for variational forms". In: *ACM Trans. Math. Softw.* 32.3, pp. 417–444. ISSN: 0098-3500. DOI: 10.1145/1163641.1163644.

Kirk, D. and W.-m. Hwu (Jan. 2010). *Programming Massively Parallel Processors*. Morgan Kaufmann.

Klöckner, A. (2014). "Loo.Py: Transformation-Based Code Generation for GPUs and CPUs". In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY'14. Edinburgh, United Kingdom: Association for Computing Machinery, pp. 82–87. ISBN: 9781450329378. DOI: 10.1145/2627373.2627387.

Klöckner, A., N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih (2012). "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation". In: *Parallel Computing* 38.3, pp. 157–174. ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.09.001.

Klöckner, A., L. C. Wilcox, and T. Warburton (2016). "Array Program Transformation with Loo.Py by Example: High-Order Finite Elements". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY 2016. Santa Barbara, CA, USA: Association for Computing Machinery, pp. 9–16. ISBN: 9781450343848. DOI: 10.1145/2935323.2935325.

Knepley, M. G. and A. R. Terrel (2013). "Finite Element Integration on GPUs". In: *ACM Transactions on Mathematical Software* 39.2. DOI: 10.1145/2427023.2427027.

Kreutzer, M., G. Hager, G. Wellein, H. Fehske, and A. R. Bishop (2014). "A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units". In: *SIAM Journal on Scientific Computing* 36.5, pp. 401–423. ISSN: 1064-8275.

Kronbichler, M. and K. Kormann (June 2012). "A generic interface for parallel cell-based finite element operator application". In: *Computers and Fluids* 63, pp. 135–147. ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2012.04.012.

Kronbichler, M. and K. Ljungkvist (May 2019). "Multigrid for Matrix-Free High-Order Finite Element Computations on Graphics Processors". In: *ACM Trans. Parallel Comput.* 6.1. ISSN: 2329-4949. DOI: 10.1145/3322813.

Langguth, J., N. Wu, J. Chai, and X. Cai (2015a). "Parallel performance modeling of irregular applications in cell-centered finite volume methods over unstructured tetrahedral meshes". In: *Journal of Parallel and Distributed Computing* 76, pp. 120–131. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.10.005.

Langguth, J., M. Sourouri, G. T. Lines, S. B. Baden, and X. Cai (July 2015b). "Scalable Heterogeneous CPU-GPU Computations for Unstructured Tetrahedral Meshes". In: *IEEE Micro* 35.4, pp. 6–15. ISSN: 0272-1732. DOI: 10.1109/MM.2015.70.

Langtangen, H. P. and A. Logg (2017). *Solving PDEs in Python*. Springer. ISBN: 978-3-319-52461-0. DOI: 10.1007/978-3-319-52462-7.

Li, R. and Y. Saad (2013). "GPU-accelerated preconditioned iterative linear solvers". In: *The Journal of Supercomputing* 63, pp. 443–466. ISSN: 0920-8542. DOI: 10.1007/s11227-012-0825-3.

Liu, H., B. Yang, and Z. Chen (2015). "Accelerating algebraic multigrid solvers on NVIDIA GPUs". In: *Computers & Mathematics with Applications* 70.5, pp. 1162–1181. ISSN: 0898-1221. DOI: 10.1016/j.camwa.2015.07.005.

Liu, W. and B. Vinter (2015). "CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication". In: *Proceedings of the 29th ACM on international conference on supercomputing*. ICS '15. ACM, pp. 339–350. ISBN: 978-1-4503-3559-1.

Liu, X., M. Smelyanskiy, E. Chow, and P. Dubey (2013). "Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors". In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ICS '13. ACM, pp. 273–282. ISBN: 978-1-4503-2130-3. DOI: 10.1145/2464996.2465013.

Ljungkvist, K. (2014). "Matrix-Free Finite-Element Operator Application on Graphics Processing Units". In: *Euro-Par 2014: Parallel Processing Workshops*. Ed. by L. Lopes, J. Žilinskas, A. Costan, R. G. Cascella, G. Kecskemeti, E. Jeannot, M. Cannataro, L. Ricci, S. Benkner, S. Petit, V. Scarano, J. Gracia, S. Hunold, S. L. Scott, S. Lankes, C. Lengauer, J. Carretero, J. Breitbart, and M. Alexander. Springer International Publishing, pp. 450–461. ISBN: 978-3-319-14313-2. DOI: 10.1007/978-3-319-14313-2_38.

Logg, A., K.-A. Mardal, G. N. Wells, et al. (2012). *Automated Solution of Differential Equations by the Finite Element Method*. Berlin: Springer. ISBN: 978-3-642-23098-1. DOI: 10.1007/978-3-642-23099-8.

Luporini, F., D. A. Ham, and P. H. J. Kelly (Mar. 2017). "An Algorithm for the Optimization of Finite Element Integration Loops". In: *ACM Transactions on Mathematical Software* 44.1. ISSN: 0098-3500. DOI: 10.1145/3054944.

Luporini, F., A. L. Varbanescu, F. Rathgeber, G.-T. Bercea, J. Ramanujam, D. A. Ham, and P. H. J. Kelly (Jan. 2015). "Cross-Loop Optimization of Arithmetic Intensity for Finite Element Local Assembly". In: *ACM Transactions on Architecture and Code Optimization* 11.4. ISSN: 1544-3566. DOI: 10.1145/2687415.

Marciniak, M., H. Arevalo, J. Tfelt-Hansen, T. Jespersen, R. Jabbari, C. Glinge, K. A. Ahtarovski, N. Vejlstrup, T. Engstrom, M. M. Maleckar, and K. McLeod (Jan. 2017). "From CMR Image to Patient-Specific Simulation and Population-Based Analysis:

Tutorial for an Openly Available Image-Processing Pipeline". In: *STACOM 2016: Statistical Atlases and Computational Models of the Heart. Imaging and Modelling Challenges*. Ed. by T. Mansi, K. McLeod, M. Pop, K. Rhode, M. Sermesant, and A. Young. Springer International Publishing, pp. 106–117. ISBN: 978-3-319-52718-5. DOI: 10.1007/978-3-319-52718-5_12.

Markall, G. R., A. Slemmer, D. A. Ham, P. H. J. Kelly, C. D. Cantwell, and S. J. Sherwin (Jan. 2013). "Finite element assembly strategies on multi-core and many-core architectures". In: *International Journal for Numerical Methods in Fluids* 71.1, pp. 80–97. ISSN: 0271-2091. DOI: 10.1002/fld.3648.

McCalpin, J. D. (Dec. 1995). "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25.

Naumov, M., M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka (2015). "AmgX: A Library for GPU Accelerated Algebraic Multigrid and Preconditioned Iterative Methods". In: *SIAM Journal on Scientific Computing* 37.5, S602–S626. DOI: 10.1137/140980260.

Nishtala, R., R. W. Vuduc, J. W. Demmel, and K. A. Yelick (May 2007). "When cache blocking of sparse matrix vector multiply works and why". In: *Applicable Algebra in Engineering, Communication and Computing* 18.3, pp. 297–311. ISSN: 1432-0622. DOI: 10.1007/s00200-007-0038-9.

NVIDIA Corporation (Aug. 2017). *NVIDIA Tesla V100 GPU architecture*. NVIDIA Corporation. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

— (July 2019). *NVRTC – CUDA runtime compilation user guide*. NVIDIA Corporation. URL: https://docs.nvidia.com/cuda/nvrtc/.

— (Aug. 2020a). *CUDA C++ Best Practices Guide*. NVIDIA Corporation. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

— (Aug. 2020b). *CUDA C++ Programming Guide*. NVIDIA Corporation. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

— (July 2020c). *cuSPARSE Library*. NVIDIA Corporation. URL: https://docs.nvidia.com/cuda/cusparse/index.html.

Ølgaard, K. B. and G. N. Wells (Jan. 2010). "Optimizations for quadrature representations of finite element tensors through automated code generation". In: *ACM Transactions on Mathematical Software* 37.1. ISSN: 0098-3500. DOI: 10.1145/1644001.1644009.

— (2012). "Applications in solid mechanics". In: *Automated Solution of Differential Equations by the Finite Element Method*. Ed. by A. Logg, K.-A. Mardal, and G. N. Wells. Berlin: Springer. Chap. 26, pp. 505–526. ISBN: 978-3-642-23098-1. DOI: 10.1007/978-3-642-23099-8.

Oliker, L., X. Li, P. Husbands, and R. Biswas (2002). "Effects of Ordering Strategies and Programming Paradigms on Sparse Matrix Computations". In: *SIAM Review* 44.3, pp. 373–393. DOI: 10.1137/S00361445003820.

Owens, J. D., D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell (2007). "A Survey of General-Purpose Computation on Graphics Hardware". In: *Computer Graphics Forum* 26.1, pp. 80–113.

Pichel, J. C., D. B. Heras, J. C. Cabaleiro, and F. F. Rivera (2005). "Performance optimization of irregular codes based on the combination of reordering and blocking techniques". In: *Parallel Computing* 31 (8–9), pp. 858–876. DOI: 10.1016/j.parco.2005.04.012.

Pichel, J. C., F. F. Rivera, M. Fernández, and A. Rodríguez (2012). "Optimization of sparse matrix–vector multiplication using reordering techniques on GPUs". In: *Microprocessors and Microsystems* 36.2, pp. 65–77. ISSN: 0141-9331. DOI: 10.1016/j.micpro.2011.05.005.

Pichler, F. and G. Haase (Mar. 2017). "Finite element method completely implemented for graphic processor units using parallel algorithm libraries". In: *The International Journal of High Performance Computing Applications* 33.1, pp. 53–66. DOI: 10.1177/1094342017694703.

Pinar, A. and M. T. Heath (1999). "Improving Performance of Sparse Matrix-vector Multiplication". In: *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*. SC '99. Portland, Oregon, USA: ACM. ISBN: 1-58113-091-0. DOI: 10.1145/331532.331562.

Rathgeber, F., D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly (Dec. 2016). "Firedrake: Automating the Finite Element Method by Composing Abstractions". In: *ACM Transactions on Mathematical Software* 43.3. ISSN: 0098-3500. DOI: 10.1145/2998441.

Reguly, I. Z. and M. B. Giles (Apr. 2015). "Finite Element Algorithms and Data Structures on Graphical Processing Units". In: *International Journal of Parallel Programming* 43.2, pp. 203–239. DOI: 10.1007/s10766-013-0301-6.

Rognes, M. E., R. C. Kirby, and A. Logg (Nov. 2009). "Efficient Assembly of $H(\mathrm{div})$ and $H(\mathrm{curl})$ Conforming Finite Elements". In: *SIAM Journal on Scientific Computing* 31.6, pp. 4130–4151. ISSN: 1064-8275. DOI: 10.1137/08073901X.

Russell, F. P. and P. H. J. Kelly (July 2013). "Optimized Code Generation for Finite Element Local Assembly Using Symbolic Manipulation". In: *ACM Transactions on Mathematical Software* 39.4. ISSN: 0098-3500. DOI: 10.1145/2491491.2491496.

Saad, Y. (2003). *Iterative methods for sparse linear systems*. 2nd ed. SIAM. ISBN: 978-0-898715-34-7.

Sourouri, M., S. B. Baden, and X. Cai (2017). "Panda: A Compiler Framework for Concurrent CPU+GPU Execution of 3D Stencil Computations on GPU-accelerated Supercomputers". In: *Int. J. Parallel Prog.* 45 (3), pp. 711–729. DOI: 10.1007/s10766-016-0454-1.

Sun, T., L. Mitchell, K. Kulkarni, A. Klöckner, D. A. Ham, and P. H. Kelly (July 2020). "A study of vectorization for matrix-free finite element methods". In: *The International Journal of High Performance Computing Applications*. DOI: 10.1177/1094342020945005.

Tang, Y., R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson (2011). "The Pochoir Stencil Compiler". In: *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '11. San Jose, California, USA: Association for Computing Machinery, pp. 117–128. ISBN: 9781450307437. DOI: 10.1145/1989493.1989508.

Temam, O. and W. Jalby (1992). "Characterizing the Behavior of Sparse Algorithms on Caches". In: *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*.

Supercomputing '92. Minneapolis, Minnesota, USA: IEEE Computer Society Press, pp. 578–587. ISBN: 0-8186-2630-5.

Tomov, S., J. Dongarra, and M. Baboulin (June 2010). "Towards dense linear algebra for hybrid GPU accelerated manycore systems". In: *Parallel Computing* 36.5-6, pp. 232–240. ISSN: 0167-8191. DOI: 10.1016/j.parco.2009.12.005.

Uhlig, R. A. and T. N. Mudge (June 1997). "Trace-driven Memory Simulation: A Survey". In: *ACM Computing Surveys* 29.2, pp. 128–170. ISSN: 0360-0300. DOI: 10.1145/254180.254184.

Unat, D., X. Cai, and S. B. Baden (2011). "Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C". In: *Proceedings of the International Conference on Supercomputing*. ICS '11. Tucson, Arizona, USA: Association for Computing Machinery, pp. 214–224. ISBN: 9781450301022. DOI: 10.1145/1995896.1995932.

Unat, D., J. Zhou, C. Y., S. B. Baden, and X. Cai (2012). "Accelerating a 3D Finite-Difference Earthquake Simulation with a C-to-CUDA Translator". In: *Computing in Science & Engineering* 14.3, pp. 48–59. DOI: 10.1109/MCSE.2012.44.

Vos, P. E., S. J. Sherwin, and R. M. Kirby (2010). "From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low- and high-order discretisations". In: *Journal of Computational Physics* 229.13, pp. 5161–5181. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2010.03.031.

Vuduc, R. W., J. W. Demmel, and K. A. Yelick (Jan. 2005). "OSKI: A library of automatically tuned sparse matrix kernels". In: *Journal of Physics: Conference Series*. Vol. 16. 1. IOP Publishing, pp. 521–530.

Vuduc, R. W., J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee (2002). "Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply". In: *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. SC '02. Baltimore, Maryland: IEEE Computer Society Press, pp. 1–35. ISBN: 0-7695-1524-X.

Willcock, J. and A. Lumsdaine (2006). "Accelerating Sparse Matrix Computations via Data Compression". In: *Proceedings of the 20th Annual International Conference on Supercomputing*. ICS '06. Cairns, Queensland, Australia: ACM, pp. 307–316. ISBN: 1-59593-282-8. DOI: 10.1145/1183401.1183444.

Williams, S., L. Oliker, R. W. Vuduc, J. Shalf, K. Yelick, and J. Demmel (2009a). "Optimization of sparse matrix-vector multiplication on emerging multicore platforms". In: *Parallel Computing* 35.3, pp. 178–194. ISSN: 0167-8191. DOI: 10.1016/j.parco.2008.12.006.

Williams, S., A. Waterman, and D. Patterson (Apr. 2009b). "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Communications of the ACM* 52.4, pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785.

Yzelman, A. N. and R. H. Bisseling (2009). "Cache-Oblivious Sparse Matrix-Vector Multiplication by Using Sparse Matrix Partitioning Methods". In: *SIAM Journal on Scientific Computing* 31.4, pp. 3128–3154. DOI: 10.1137/080733243.

— (2012). "A Cache-Oblivious Sparse Matrix–Vector Multiplication Scheme Based on the Hilbert Curve". In: *Progress in Industrial Mathematics at ECMI 2010*. Ed. by M. Günther, A. Bartel, M. Brunk, S. Schöps, and M. Striebel. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 627–633. ISBN: 978-3-642-25100-9.

Zhang, Y. and F. Mueller (2012). "Auto-Generation and Auto-Tuning of 3D Stencil Codes on GPU Clusters". In: *Proceedings of the Tenth International Symposium on*

Code Generation and Optimization. CGO '12. San Jose, California: Association for Computing Machinery, pp. 155–164. ISBN: 9781450312066. DOI: 10.1145/2259016.2259037.

Papers

Paper I

Cache simulation for irregular memory traffic on multi-core CPUs: case study on performance models for sparse matrix-vector multiplication

James D. Trotter, Simula Research Laboratory and University of Oslo, Norway
Johannes Langguth, Simula Research Laboratory
Xing Cai, Simula Research Laboratory and University of Oslo, Norway

Published in *Journal of Parallel and Distributed Computing*, Volume 144, October 2020, pages 189–205. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2020.05.020.

Abstract

Parallel computations with irregular memory access patterns are often limited by the memory subsystems of multi-core CPUs, though it can be difficult to pinpoint and quantify performance bottlenecks precisely. We present a method for estimating volumes of data traffic caused by irregular, parallel computations on multi-core CPUs with memory hierarchies containing both private and shared caches. Further, we describe a performance model based on these estimates that applies to bandwidth-limited computations. As a case study, we consider two standard algorithms for sparse matrix-vector multiplication, a widely used, irregular kernel. Using three different multi-core CPU systems and a set of matrices that induce a range of irregular memory access patterns, we demonstrate that our cache simulation combined with the proposed performance model accurately quantifies performance bottlenecks that would not be detected using standard best- or worst case estimates of the data traffic volume.

1 Introduction

Performance is a high priority in scientific computations, and so meticulous work is devoted to optimising the underlying code. During such optimisation efforts, performance models are valuable tools for directing attention towards pressure points, and indicating when optimisations are good enough and expending further effort would be unproductive. For instance, the popular Roofline model [Williams et al. 2009b] bounds performance in terms of a CPU's peak computational capacity and memory bandwidth together with an algorithm's computational intensity. Because CPUs have hierarchical memories, the bandwidth and computational intensity can vary depending on the memory hierarchy level that is considered. Moreover, the computational intensity depends not only on parameters such as cache size, but also on the memory access pattern of the computation. Recently, more elaborate performance models have been developed for stencil codes [Cruz and Araya-Polo 2015; Stengel et al. 2015; Zhang and Cai 2016], where they have been used to evaluate the effectiveness of spatial

and temporal blocking optimisations. In these cases, the amounts of data transferred between levels of the memory hierarchy is known in advance, because memory accesses are predictable and depend only on the problem size and the order of the stencil. Unfortunately, this is not the case for irregular computations, where memory access patterns depend on data that may only be known at runtime.

When faced with irregular access patterns, the typical approach is to derive estimates of the memory traffic for the worst- or best case scenarios. These are "paper and pencil" estimates that have the advantage of being cheap to produce, not requiring any implementation or actual machine to run. On the other hand, such estimates are crude and can in reality be far from the true data traffic volumes, thereby rendering little help in understanding the actual performance that is achieved. For example, Fig. 1 shows worst- and best case estimates for sparse matrix-vector multiplication (SpMV), a widely used computational kernel that suffers from both irregularity and low computational intensity. Due to the considerable difference between the best- and worst case data traffic, these estimates cannot provide much confidence if they are used to evaluate whether the performance of a given kernel implementation is good enough. In this case, more accurate estimation of data traffic volumes is needed for performance validation. In general, numerous computational kernels face the same issues due to irregular memory accesses that arise through the use of sparse data structures, such as graphs or unstructured meshes.

In this paper, we present a method for quantifying the amounts of data transferred between levels of a multi-core CPU's memory hierarchy during irregular computations. The estimated data traffic volumes are produced by a trace-driven cache simulation that relies on a few basic assumptions and a simplified model of the memory hierarchy. Moreover, the method applies to memory hierarchies with shared caches, a common feature of contemporary multi-core CPUs, and a case that is not always addressed by existing analytical cache models [Agarwal et al. 1989; Frigo et al. 2012]. Because the proposed method is based on tracing a sequence of memory references, it requires some amount of computation that is likely to be at least as much as the cost of executing the kernel itself. However, the method remains applicable in cases where the actual machine in question is not available, or the data traffic cannot be quantified directly through hardware monitoring facilities, for example, because these facilities are unavailable, unreliable or the results are not easily interpreted.

Because of its importance and familiarity as an irregular computational kernel, we use SpMV to demonstrate that our cache simulation accurately quantifies the volumes of data transfers in the memory hierarchies of two Intel-based multi-core CPU systems. In turn, these data transfer volumes are used to give accurate performance predictions that are unavailable through the use of simple worst- or best case estimates. We also give performance predictions for an AMD Epyc CPU, and explore some limitations of the proposed method using a variant of SpMV that not only includes irregular reads, but also irregular writes. Ultimately, these predictions result in a quantitative understanding of SpMV performance, which, for example, can be used to check that the observed performance of a given implementation matches with our expectations, and that the implementation is free from hidden performance issues.

The remainder of this paper is organised as follows. In the next section, we describe our cache simulation approach for estimating the data traffic volumes of computations

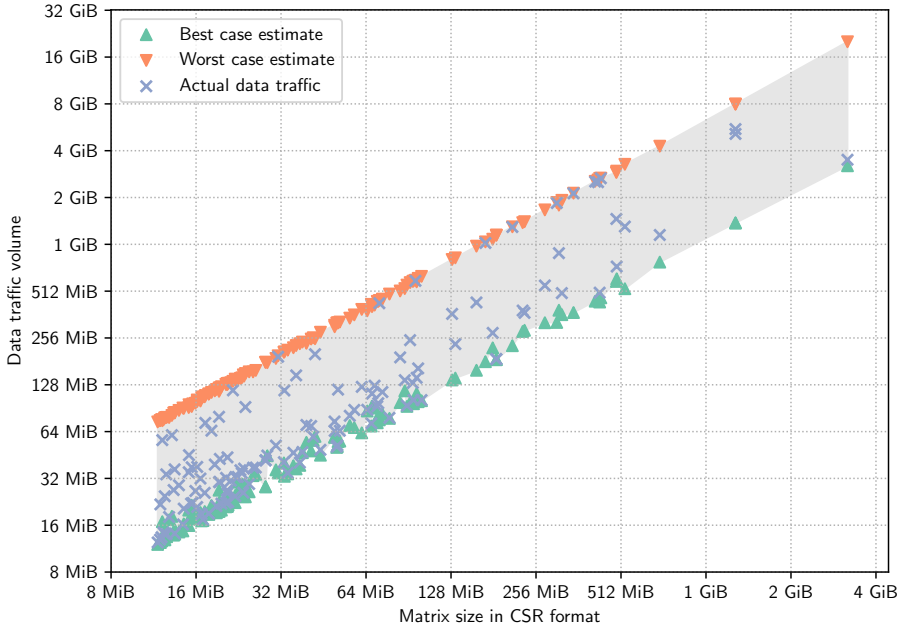


Figure 1. Worst- and best case data traffic volumes for sparse matrix-vector multiplication. The data traffic estimates (Eqs. (5) and (6)) are based on a standard algorithm (see Section 4) and are shown for matrices from the SuiteSparse Matrix Collection [Davis and Hu 2011]. For comparison, actual data traffic volumes are shown based on the measured number of cache misses for the 32 KiB L1 data cache with 64 byte cache lines on an Intel Sandy Bridge Xeon E5-2650 CPU. (See Section 5.3 for details on how these measurements were obtained.) Notice that there is a wide gap between the best- and worst case data traffic volumes. For a given sparse matrix, the actual volume of data traffic depends on the size of the CPU cache and the sparsity pattern of the matrix.

with irregular memory accesses. In Section 3, we present a performance model for bandwidth-limited computations, where the relevant data traffic volumes are used together with realistic memory and CPU cache bandwidths. Next, in Section 4, we recall standard SpMV algorithms for matrices in the compressed sparse row (CSR) and coordinate (COO) storage formats. We also review known bounds on the volume of data traffic generated by the CSR SpMV algorithm, which is later used to compare with the results of our cache simulation method. Then, in Section 5, we describe experiments that are used to validate the estimated data traffic volumes and the performance model for the studied SpMV algorithms. Finally, we briefly discuss related work in Section 6 and draw our conclusions in Section 7.

2 Quantifying data traffic for irregular, parallel computations

To estimate the data traffic volume for a given computation on a multi-core CPU system, we consider the sequence of load and store operations that would be performed by the participating CPUs. Then we simulate a cache's behaviour using a simplified model based on the established *ideal-cache model* [Frigo et al. 2012], which is ordinarily used in the design of cache-oblivious algorithms. We depart from the ideal-cache model in two ways. First, for practical reasons, we assume a *least recently used* replacement policy, instead of the ideal-cache model's optimal policy, also known as Bélády's algorithm. Second, we extend our cache model to incorporate multi-level memory hierarchies and caches that are shared by multiple processors.

2.1 A simplified cache model

Consider the case of a sequential computation in a memory hierarchy consisting of main memory and a single CPU cache. More specifically, consider a cache of size Z that is partitioned into cache lines of size L, such that there are Z/L cache lines. Data is moved between main memory and the cache in contiguous blocks of size L, called cache blocks. These transfers originate from load or store operations issued by the CPU.

Furthermore, we make the following three assumptions regarding the behaviour of the cache:

- First, we assume that the cache operates with demand caching, which means that a cache block is placed into the cache only when the CPU issues a load or store referencing a memory address corresponding to that cache block. This assumption precludes the use of hardware prefetching, a common feature in multi-core CPUs that is used to speculatively fetch data before it is requested. However, to allow hardware prefetching would complicate our cache simulation in ways that are most likely specific to particular hardware.

- Second, the cache is assumed to be fully associative, meaning that any cache block can be placed into any cache line. In this way, we disregard conflict misses, that is, cache misses due to cache blocks that would be mapped to the same set in a set-associative or direct-mapped cache. The mapping of cache blocks to sets is hardware-specific, as well as documented poorly or not at all. Moreover, the overall effect of conflict misses is likely to be small and to only have an impact on a few, very specific memory access patterns.

- Third, we assume that cache lines are evicted according to a least recently used policy whenever the cache is full. This policy can be easily simulated, and it also seems to approximate real hardware accurately enough for our purposes.

The volume of data traffic between the cache and main memory is deduced by counting transfers of cache blocks that result from a given sequence of loads and stores. The procedure is described in the following and summarised in Algorithm 1. Throughout the cache simulation, we maintain a *least recently used list* with Z/L

```
Q ← 0
Initialise list as empty
for each load or store address α do
  if ⌊α/L⌋ is in list then
    move ⌊α/L⌋ to front of list
  else
    Q ← Q + 1
    if list has Z/L entries then
      remove last item from list
    end if
    insert ⌊α/L⌋ at the front of list
  end if
end for
```

Algorithm 1. A CPU cache simulation for a fully associative cache of size Z with cache lines of size L using a least recently used replacement policy. The number of cache misses is denoted by Q.

entries, one per cache line, to keep track of the cache blocks that occupy the cache. The list is also used to decide which cache line to evict, if the cache is full and more room is needed for incoming cache blocks. Initially, the list is empty, thereby representing a cold cache that does not contain any relevant data prior to the computation. Next, each load or store references a memory address, α, and a corresponding block of main memory, indexed by $\lfloor \alpha/L \rfloor$. If the referenced cache block is found in the least recently used list, then it is moved to the front of the list. Otherwise, a cache miss occurs. In this case, the number of cache misses is increased by one, and, if the least recently used list is full, the last entry of the list is removed. Finally, the referenced cache block is added to the front of the list. Thus, the total number of cache misses is equal to the number of new entries that were inserted at the front of the least recently used list.

2.2 Multi-level hierarchies, private and shared caches

In the case of a memory hierarchy with multiple cache levels, we consider the CPU's working memory to be the highest level of the memory hierarchy, whereas main memory is the lowest level. Caches that lie near the CPU are referred to as higher-level caches, and, conversely, caches farther from the CPU are lower-level caches. Furthermore, we assume that each cache is inclusive of any higher-level caches, which means that cache blocks contained in a higher-level cache are also contained in lower-level caches. For example, data that is brought to a first-level (L1) cache must also be placed into a last-level cache. With this assumption, each cache can be treated individually. Letting Z_i denote the size of the i-th cache and L the size of blocks that are transferred from the next, lower level of the memory hierarchy, we obtain the corresponding number of cache misses Q_i by using a least recently used list with Z_i/L entries and following Algorithm 1.

Next, consider a parallel computation with multiple CPUs, where each CPU has a

private cache. For the private cache belonging to the p-th processor, let $Z^{(p)}$ and $L^{(p)}$ denote the cache size and cache line size, respectively. This case is again handled by applying Algorithm 1 to each cache individually. However, only the loads and stores that are issued by the p-th processor are used to count the number of cache misses $Q^{(p)}$ for the corresponding private cache.

Now, consider a parallel computation with multiple CPUs sharing a cache. Generally speaking, each CPU issues its load and store instructions independently of the others. Therefore, the order in which a shared cache receives data requests is affected by factors such as the scheduling of threads onto CPUs. Moreover, if CPUs retrieve data from different levels of the memory hierarchy, they might proceed at different speeds because of varying memory access latencies—that is, the time it takes to serve each request depends on where in the memory hierarchy the data is located. Ultimately, the order of data requests to a shared cache, and thus the number of cache misses, may differ between runs of the same, identical program due to competition between the CPUs.

However, we propose to approximate the order of data requests to a shared cache by assuming that requests are submitted by processors in a round-robin fashion. In turn, each CPU issues its next load or store operation, resulting in an interleaving of the data requests from each CPU to the shared cache. The number of cache misses for each CPU is obtained from the interleaved sequence of loads and stores by applying Algorithm 1 with the following modification: Rather than estimating only a total number of cache misses, we maintain a counter $Q^{(p)}$ for each CPU, which is incremented whenever a cache miss is caused by a load or store that was issued by the p-th processor. The total number of cache misses is $Q = \sum_p Q^{(p)}$. Attributing data traffic to individual CPUs will be important when these data traffic estimates are tied to the overall performance in Section 3.

Finally, for systems with a non-uniform memory access (NUMA) architecture, it may be necessary to distinguish between multiple memory controllers or channels that may serve data in response to last-level cache misses. In this case, memory addresses are partitioned into NUMA domains according to some memory policy, and separate counters are used for the cache misses for each NUMA domain.

2.3 Limitations

We have already mentioned some limitations of the proposed cache simulation method. For instance, we ignore conflict misses and hardware prefetching, and, in the case of a shared cache, we assume a simple round-robin scheduling of requests from participating CPUs. In addition, the proposed method treats stores in the same way as loads and ignores any additional data transfers that may be required to maintain cache coherence for multi-core CPUs with private caches. In practice, a store will typically result in two data transfers. First, the relevant cache block is transferred from a lower-level cache or main memory before the store operation is carried out in the working memory of the CPU. Second, the modified cache line must be written back to lower-level caches and, eventually, to main memory. However, our proposed cache simulation method does not account for traffic resulting from writing back modified cache lines, or additional traffic arising from write-related cache coherency effects,

such as false sharing. For kernels whose performance is significantly impacted by any of the above-mentioned effects, the current approach would have to be extended or improved.

3 A performance model based on data traffic and bandwidth

In this section, we describe a performance model for computations that are limited by cache or memory bandwidth by tying the execution time to relevant data traffic volumes between levels inside a memory hierarchy. In the following model, we assume that computation and memory accesses overlap, and, moreover, that the dominant cost is due to memory accesses, so that computations can be neglected. In addition, data must be transferred between adjacent memory hierarchy levels, and we assume that these transfers occur in parallel.

For the i-th cache in a multi-level hierarchy, recall that L is the cache line size and let $Q_i^{(p)}$ be the number of cache misses for the p-th processor that is attached to the i-th cache. Then $V_i^{(p)} = L \cdot Q_i^{(p)}$ is the data traffic volume that is transferred to the i-th cache from lower levels of the memory hierarchy on account of the p-th processor. Furthermore, let λ_i denote the sustainable, single-core bandwidth for transferring the data, and let μ_i denote the aggregate bandwidth that is available to all the CPU cores that are sharing the i-th cache. Although hardware vendors occasionally specify theoretical bandwidth numbers, these are rarely achievable in practice. Instead, realistic, sustainable bandwidths are typically determined by a benchmark program, such as STREAM [McCalpin 1995].

First, performance may be limited due to data traffic induced by individual CPU cores. In this case, a lower bound on the execution time T is given by the per-core data transfer that takes the longest time, that is,

$$T \geq \max_{i,p} \frac{V_i^{(p)}}{\lambda_i}, \qquad \text{or} \qquad S \leq \min_{i,p} \frac{F\lambda_i}{V_i^{(p)}}, \tag{1}$$

where F is the number of floating-point operations performed, and $S = F/T$ is the performance in floating-point operations per unit time. In other words, a lower bound on the execution time T translates to an equivalent upper bound on performance.

Second, whenever resources in a memory hierarchy are shared by multiple CPU cores, performance may be limited due to the total data traffic volume and a limited, aggregate bandwidth,

$$T \geq \max_{i} \frac{\sum_p V_i^{(p)}}{\mu_i}, \qquad S \leq \min_{i} \frac{F\mu_i}{\sum_p V_i^{(p)}}. \tag{2}$$

In particular, for data transferred between main memory and last-level caches, the aggregate bandwidth does not scale linearly with the number of CPU cores, but instead saturates before all the cores are in use. Moreover, many multi-core CPUs have a NUMA architecture, where main memory is partitioned into NUMA domains, and the bandwidth of each NUMA domain is limited. In some cases, it is also necessary

to take into account the fact that groups of CPU cores have an affinity to their local NUMA domain. This is especially the case for problems where a significant amount of data is transferred between CPU cores and their remote NUMA domains, since the bandwidth of such transfers tends to be severely limited.

4 Sparse matrix-vector multiplication

The multiplication of a sparse matrix with a dense vector, or SpMV, is a prime example of an irregular, parallel computation. It is also a fundamental computational kernel that appears in numerous scientific applications. For example, SpMV is performed repeatedly in iterative methods for solving sparse linear systems, such as Krylov subspace methods [Saad 2003]. The efficiency of these methods often hinges on the SpMV computations that are required during each iteration, but it is well known that SpMV has a low computational intensity, and, therefore, its performance lies well below peak computational capacity [Toledo 1997; Vuduc 2004]. More precisely, every non-zero value in a sparse matrix gives rise to one multiplication and one addition, but typically requires loading at least two floating-point numerical values and one integer index. In the general case, the locations of the matrix non-zeros are represented explicitly, which means that half of the floating-point values are read indirectly, indexed by the column indices of the matrix non-zeros. These indirect reads can induce a highly irregular memory access pattern, which prevents data reuse in the caches. In any case, performance is typically limited by the speed at which data can be retrieved from CPU caches or main memory, rather than the peak floating-point capacity. But the relationship between the locations of matrix non-zeros, also called the sparsity pattern, and the amount of data that must be read from each level of the memory hierarchy is difficult to characterise. In other words, for a given matrix, it is not obvious which level of the memory hierarchy constitutes a performance bottleneck.

There is a vast literature devoted to optimising SpMV performance, including low-level code optimisations [Williams et al. 2009a], and optimisations for reducing the volume of data traffic, such as register blocking [Pinar and Heath 1999; Vuduc et al. 2002], cache blocking [Nishtala et al. 2007], matrix re-orderings [Heras et al. 2001; Yzelman and Bisseling 2011], and compression [Willcock and Lumsdaine 2006]. In addition, advanced SpMV algorithms have been proposed, either based on the standard compressed sparse row format and variants thereof [Buluç et al. 2009; W. Liu and Vinter 2015; Merrill and Garland 2016; Yzelman and Bisseling 2012] or other sparse matrix formats [Haase et al. 2007; Kreutzer et al. 2014; X. Liu et al. 2013]. Furthermore, sparse matrix partitioning schemes [Akbudak et al. 2013; Çatalyürek and Aykanat 1999; Karsavuran et al. 2016; Vastenhouw and Bisseling 2005; Yzelman and Roose 2014] are used to improve the parallel efficiency of SpMV through better load balancing and reduced communication. Although there has recently been a considerable effort to optimise SpMV for GPUs [Filippone et al. 2017], we focus our attention on multi-core CPUs, which still represent a relevant and important case.

Note, however, that we do not propose any new optimisations or storage formats for sparse matrices. Instead, we aim to understand the performance of irregular, bandwidth-limited computations, by using a pair of simple, well known SpMV algo-

rithms as examples.

4.1 Compressed Sparse Row

Compressed sparse row (CSR) is a standard storage format for general sparse matrices, where the non-zero matrix entries and their locations are stored explicitly. Consider a sparse matrix $A = (a_{i,j}) \in \mathbf{R}^{m,n}$ with m rows, n columns, and N non-zeros. In other words, there are N distinct pairs of row and column indices, (i_k, j_k), such that $a_k := a_{i_k,j_k} \neq 0$. If the non-zeros are arranged in ascending order according to their row indices, then the matrix A can be represented in the CSR format with N non-zero values $(a_0, a_1, \ldots, a_{N-1})$ and column indices $(j_0, j_1, \ldots, j_{N-1})$, and $(m+1)$ row pointers $(r_0, r_1, \ldots, r_m)$. The row pointers r_i and $r_{i+1} - 1$ are the indices of the first and last non-zeros of the i-th row, respectively. That is, a non-zero a_k belongs to row i_k if $r_{i_k} \leq k < r_{i_k+1}$.

For a source vector $x \in \mathbf{R}^n$ and destination vector $y \in \mathbf{R}^m$, the SpMV $y \leftarrow y + Ax$ is defined as

$$y_i \leftarrow y_i + \sum_{k=r_i}^{r_{i+1}-1} a_k x_{j_k}, \qquad \text{for } i = 0, 1, \ldots, m-1. \tag{3}$$

A common strategy for computing SpMV in parallel is to partition the rows of the matrix and distribute the parts among multiple processors or threads. In this way, each processor computes its destination vector values y_i according to the row partitioning. The non-zero values a_k, column indices j_k, and destination vector elements y_i are partitioned along with the rows. In contrast, the source vector elements x_{j_k} may be shared by several processors, though the extent of this sharing depends on the column indices j_k.

The code in Algorithm 2 shows a straightforward C implementation of shared-memory parallel SpMV for a matrix in the CSR format. Here, OpenMP is used to parallelise the outer loop by distributing the rows among a given number of threads. The `schedule` clause of the `omp for` compiler directive specifies that the loop is to be divided into chunks, consisting of *chunk_size* consecutive loop iterations, and that the chunks are assigned to threads in a round-robin fashion. Unless otherwise noted, we omit the *chunk_size*, which implies that the threads are assigned one large chunk each.

4.2 Coordinate storage format

The coordinate (COO) format is another scheme for storing sparse matrices, though it is not commonly used for SpMV due to its poor performance. The main reason is that COO does not compress the row- and column indices, which leads to a larger memory footprint and more memory traffic during an SpMV calculation compared to CSR. However, a COO-based SpMV algorithm is interesting because it features irregular writes, which are not required for a CSR-based SpMV. Therefore, we include COO-based SpMV as an additional irregular kernel that we use to investigate the accuracy of the proposed cache simulation method.

A sparse matrix A with N non-zeros is stored in the COO format by means of three arrays: the row indices $(i_0, i_1, \ldots, i_{N-1})$, column indices $(j_0, j_1, \ldots, j_{N-1})$, and

```
void csr_spmv(
  int m,
  int const * r,
  int const * j,
  double const * a,
  double const * x,
  double * y)
{
  #pragma omp for schedule(static, chunk_size)
  for (int i = 0; i < m; i++) {
    double z = 0.0;
    for (int k = r[i]; k < r[i+1]; k++)
        z += a[k] * x[j[k]];
    y[i] += z;
  }
}
```

Algorithm 2. Parallel SpMV for matrices in the CSR storage format.

non-zero values $(a_0, a_1, \ldots, a_{N-1})$. The non-zeros are not required to be sorted by their row or column indices, and may thus appear in any order. Given a source vector $x \in \mathbf{R}^n$ and destination vector $y \in \mathbf{R}^m$, the SpMV $y \leftarrow y + Ax$ is computed as

$$y_{i_k} \leftarrow y_{i_k} + a_k x_{j_k}, \qquad \text{for } k = 0, 1, \ldots, N-1. \tag{4}$$

The code in Algorithm 3 shows a naive C implementation of shared-memory parallel SpMV for a matrix in the COO format. Note that care must be taken to avoid race conditions due to the irregular writes. For this reason, an `omp atomic` directive is used within the loop. Although there is a substantial synchronisation overhead imposed by atomic writes, and there are more efficient ways of implementing COO SpMV that avoid this overhead, we deliberately choose the current version to study the effect of irregular writes to a shared array.

4.3 Best- and worst case bounds on data traffic for CSR SpMV

In this section, we recall well known bounds on the number of cache misses for the sequential version of the CSR SpMV algorithm. Let A be a sparse matrix with m rows, n columns, and N non-zeros in the CSR format. Furthermore, assume that row pointers and column indices are stored as four-byte integers, and non-zero values are stored as eight-byte double precision floats. We define the size of the matrix as the memory footprint of the CSR format, that is, $4(m+1) + 12N$ bytes. Moreover, the working set is the combined size of the matrix A, and the vectors x and y, that is, $4(m+1) + 12N + 8m + 8n$.

To determine the number of cache misses incurred by Algorithm 2, assume, for the sake of simplicity, that each of the arrays `r`, `j`, `a`, `x`, and `y` are aligned to a cache line boundary. Otherwise, there is at most one additional cache miss for each of the

```
void coo_spmv(
  int N,
  int const * i,
  int const * j,
  double const * a,
  double const * x,
  double * y)
{
  #pragma omp for schedule(static, chunk_size)
  for (int k = 0; k < N; k++) {
    #pragma omp atomic
    y[i[k]] += a[k] * x[j[k]];
  }
}
```

Algorithm 3. Parallel SpMV for matrices in the COO format.

arrays. Further, suppose that the cache does not contain any relevant data to begin with, which holds for the first application of SpMV in an iterative solver, but also during subsequent iterations provided that the working set exceeds the size of the cache under consideration. In the best case, there are only compulsory misses because data must be brought into the cache whenever it is first accessed. Consequently, a lower bound on the number of cache misses Q coincides with the number of cache lines occupied by the working set,

$$Q \geq \overbrace{\lceil 4(m+1)/L \rceil}^{r_i} + \overbrace{\lceil 8m/L \rceil}^{y_i} + \overbrace{\lceil 8N/L \rceil}^{a_k} + \overbrace{\lceil 4N/L \rceil}^{j_k} + \overbrace{\lceil 8n/L \rceil}^{x_{j_k}}, \tag{5}$$

where L is the cache line size. The lower bound is attained only if a) there are no conflict misses, that is, data is never evicted due to using a set-associative or direct-mapped cache when it would otherwise have been reused, and b) there are no capacity misses because the sparsity pattern of the matrix leads to perfect reuse of the source vector elements x_{j_k}, and, in addition, the row pointers r_i and destination vector y_i remain in the cache between iterations of the outer loop over the rows.

To obtain a reasonable upper bound, we disregard conflict misses and assume that only the source vector suffers from capacity misses. This is equivalent to the cache being fully associative and the matrix rows being short enough that row pointers and destination vector elements do not suffer capacity misses. In the worst case, every access to a source vector element x_{j_k} incurs a cache miss, and the number of cache misses Q is bounded from above by

$$Q \leq \overbrace{\lceil 4(m+1)/L \rceil}^{r_i} + \overbrace{\lceil 8m/L \rceil}^{y_i} + \overbrace{\lceil 8N/L \rceil}^{a_k} + \overbrace{\lceil 4N/L \rceil}^{j_k} + \overbrace{N}^{x_{j_k}}. \tag{6}$$

As shown in Fig. 1, the worst- and best case estimates provide only a rough idea of the actual data traffic volume $V = Q \cdot L$. More accurate estimates of Q need to take the sparsity pattern of the matrix into account, as considered in Section 4.4.

Bounds similar to those given above have been presented previously in the literature in various forms, for example, by Heras et al. [2001] for the case where the source vector x fits in cache, but irregular accesses may cause conflict misses due to using a direct-mapped or set-associative cache. Vuduc et al. [2002] consider the case where a register blocking optimisation has been applied. Bender et al. [2010] prove tight lower bounds on the number of cache misses of a sequential SpMV algorithm for large classes of general sparse matrices, and they also describe an asymptotically optimal algorithm based on cache-aware or cache-oblivious sorting that attains these lower bounds. See also Ballard et al. [2014, Section 6] for a survey regarding both sequential and parallel SpMV, where the authors consider movement of data between levels of a memory hierarchy, as well as between parallel processors connected across a network. A different approach to estimating the data traffic volume of SpMV is described by Temam and Jalby [1992] based on a probabilistic model. However, their model is primarily concerned with cache misses caused by conflicts in a direct-mapped cache, and assumes that the matrix non-zeros are uniformly distributed, which is rarely the case.

4.4 Data traffic estimates based on cache simulation

By applying the cache simulation method presented in Section 2, we obtain an alternative estimate for the number of cache misses in the CSR SpMV algorithm. To do so, we apply Algorithm 1 to the sequence of loads and stores that would be issued by Algorithm 2. There are in total $3N + 2m + 1$ loads and m stores, where m and N are the number of matrix rows and non-zeros, respectively. To break it down, the inner loop performs $3N$ loads to fetch the non-zero values, column indices and source vector elements, while the outer loop performs $m + 1$ loads for the row pointers, and m loads and m stores for the destination vector. Multi-level memory hierarchies and shared caches are handled as described in Section 2.2. In this case, if P processors are used, we assume that the p-th processor is assigned consecutive rows from $p\lceil m/P \rceil$ up to $(p + 1)\lceil m/P \rceil$.

Crucially, this method of quantifying data traffic volumes for SpMV produces estimates that inherently depend on the sparsity pattern of the matrix. In this way, our estimates more accurately capture the true volume of data traffic generated by a given SpMV computation compared to the best- and worst case estimates Eqs. (5) and (6). Moreover, our method also incorporates the cache size, and it can therefore produce different estimates for each level of a memory hierarchy. These estimates can subsequently be used to diagnose performance bottlenecks that could not be identified by previous estimates.

Finally, we observe that the above procedure may be similarly applied to the COO SpMV kernel in Algorithm 3, other SpMV algorithms, or other irregular kernels. To do so, one must deduce the relevant sequence of load and store operations, and corresponding memory addresses, for the computation of interest.

Table 1. Matrices from the SuiteSparse Matrix Collection [Davis and Hu 2011] used in our experiments. The SuiteSparse matrices are sorted according to the number of columns.

				Non-zeros per row			
Matrix	Rows	Columns	Non-zeros	Mean	Median	Std.	Max
TSOPF_RS_b2383	0.04M	0.04M	16M	424	6	484	983
spal_004	0.01M	0.3M	46M	4 525	5 063	1 492	6 029
RM07R	0.4M	0.4M	37M	98	125	69	295
relat9	12.3M	0.5M	39M	4	4	0	4
HV15R	2.0M	2.0M	283M	140	156	54	484
GL7d19	1.9M	2.0M	37M	20	20	3	121
sx-stackoverflow	2.6M	2.6M	36M	14	2	138	38 148
FullChip	3.0M	3.0M	27M	9	6	1 807	2 312 481
Freescale1	3.4M	3.4M	19M	6	5	2	27
circuit5M	5.6M	5.6M	60M	11	5	1 357	1 290 501
Hardesty3	8.2M	7.6M	40M	5	5	0	5
Lynx68*	6.8M	6.8M	112M	16	17	2	17
Lynx68_reordered*	6.8M	6.8M	112M	16	17	2	17

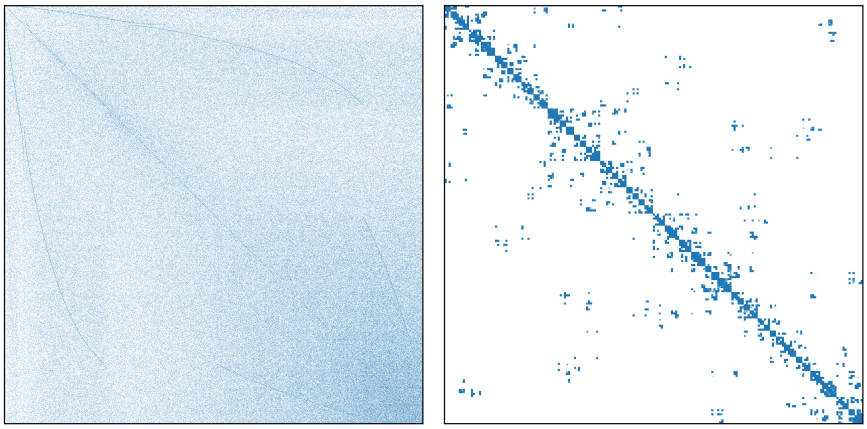
* These matrices are not found in the SuiteSparse Matrix Collection.

5 Numerical experiments

In this section, we describe experiments that test the accuracy of data traffic estimates obtained with the cache simulation method described in Section 2, focusing on the CSR-based SpMV kernel in Algorithm 2. Next, we use the data traffic estimates for CSR SpMV to evaluate the performance model from Section 3. Finally, we also evaluate the data traffic estimates for the COO SpMV kernel in Algorithm 3.

5.1 Sparse matrices

The data used in this study consists of real-valued matrices, most of which are from the SuiteSparse Matrix Collection [Davis and Hu 2011], a large set of sparse matrices from a variety of real-world applications. The selected matrices were mainly chosen because they exhibit a variety of sparsity patterns and are large enough that the SpMV working set exceeds the last-level caches of our test systems. In addition, we have included two matrices from a cardiac electrophysiology simulation that have previously been used by Langguth et al. [2015b]. Both matrices are derived from a finite volume method on the same tetrahedral mesh, but each matrix represents a different numbering of the mesh cells. Thus, the matrices have the same number of rows, columns and non-zeros, but very different sparsity patterns, see Fig. 2. For a list of the matrices used in our experiments, see Table 1, which also displays some high-level statistics about the sparsity pattern of each matrix. It is also worth noting that the matrices spal_004, GL7d19, sx-stackoverflow, and Lynx68, have particularly irregular sparsity patterns.



(a) `Lynx68` **(b)** `Lynx68_reordered`

Figure 2. The sparsity patterns of the `Lynx68` and `Lynx68_reordered` matrices. The shaded areas represent the locations of non-zero matrix entries. Both matrices have the same number of rows, columns and non-zeros, but they have very different sparsity patterns.

5.2 Experimental setup

For our experiments, we used three multi-core CPU systems: a dual-socket Intel Sandy Bridge Xeon E5-2650, a dual-socket Intel Skylake Xeon Platinum 8168, and a dual-socket AMD Epyc 7601. On all three systems, each processor core has an L1 instruction cache and L1 data cache, as well as a unified L2 cache for both instructions and data. In addition, there are L3 caches on each system, which may also contain both instructions and data. For the Intel processors, there is a shared L3 cache for each socket, whereas the AMD Epyc has an 8 MiB L3 cache for each group of four processor cores, also referred to as a compute complex. Similarly, there is a NUMA domain for each socket on the Intel systems, whereas a NUMA domain on the AMD Epyc is associated with a group of two compute complexes, or eight processor cores. Every cache uses a common cache line size of 64 bytes. The main characteristics of the hardware are summarised in Table 2. Recall that in a memory hierarchy, a lower-level cache is said to be inclusive of a higher-level cache if every cache block contained in the higher-level cache is also present in the lower-level cache. We note that the L3 cache is inclusive of the L2 cache on Sandy Bridge, but not on Skylake [Intel Corporation 2018, Ch. 2].

In the following experiments, turbo boost is disabled, and the *intel_pstate* or *acpi-cpufreq* performance scaling driver is used with the "performance" scaling governor. As a consequence, the CPU clock frequency is capped at 2.0 GHz on Sandy Bridge, 2.7 GHz on Skylake, and 2.2 GHz on Epyc, respectively. Furthermore, each thread is pinned to its own CPU core, and memory that is accessed by a given thread is initialised so that a first touch policy will place the corresponding memory pages in the NUMA domain closest to the thread.

Table 2. Multi-core CPU systems used in our experiments, including the size and set associativity of each cache, memory configuration, and theoretical memory bandwidth.

	Sandy Bridge	Skylake	Epyc
CPUs	2× Intel Xeon E5-2650	2× Intel Xeon Platinum 8168	2× AMD Epyc 7601
CPU cores	2× 8	2× 24	2× 32
L1 cache per core	32 KiB, 8-way	32 KiB, 8-way	32 KiB, 8-way
L2 cache per core	256 KiB, 8-way	1024 KiB, 16-way	512 KiB, 16-way
L3 cache per core	2560 KiB, 20-way	1408 KiB, 11-way	2048 KiB, 16-way
Memory per socket	4× 1 600 MHz	6× 2 666 MHz	8× 2 666 MHz
Memory bandwidth	2× 51.2 GB/s	2× 128.0 GB/s	2× 170.6 GB/s

Moreover, each system is running Ubuntu 18.04.2 LTS with Linux kernel version 4.15, and code is compiled using GCC 7.4.0 with the compiler flags `-O3 -fopenmp -march=native`.

As an initial verification of the SpMV kernel in Algorithm 2, Table 3 shows that the serial performance on Sandy Bridge closely matches a reference implementation provided by the Intel Math Kernel Library (MKL) 2017. In addition, we compare the performance of our implementation on both Sandy Bridge and Epyc using different compilers. For a few matrices, the Intel C compiler attains a slight performance increase, while the performance slightly degrades for other matrices. In particular, for matrices that are irregular, the difference in performance is negligible. Both because it is widely used and freely available, we will stick to using GCC for the remaining experiments.

5.3 Estimating and measuring data traffic for CSR-based SpMV

For each matrix in Table 1, we use the method described in Section 2 to estimate the number of cache misses $\hat{Q}_{\text{L1}}^{(p)}$ and $\hat{Q}_{\text{L2}}^{(p)}$ incurred by the p-th processor for the L1 and L2 caches, respectively. Because the L3 cache is shared, the corresponding estimates are produced by interleaving load and store instructions issued by each processor, as described in Section 2.2. Moreover, we obtain separate estimates for the number of L3 cache misses $\hat{Q}_{\text{L3},d}^{(p)}$ for each NUMA domain $d = 0, 1, \ldots, D-1$. We use a common cache line size of 64 bytes, and the cache sizes listed in Table 2. Based on an estimated number of cache misses per core, $Q^{(p)}$, we deduce the corresponding volume of data traffic, $\hat{V}^{(p)} = 64 \cdot \hat{Q}^{(p)}$. In the results that follow, we focus on the total data traffic volumes for each cache level, $\hat{V}_i = \sum_p \hat{V}_i^{(p)}$ for $i = \text{L1}, \text{L2}$ and $\hat{V}_{\text{L3}} = \sum_{p,d} \hat{V}_{\text{L3},d}^{(p)}$.

To validate the estimated data traffic volumes, we now compare them to data traffic measurements on our Intel-based multi-core CPUs. Unfortunately, currently available tools for hardware performance monitoring on Linux, in particular, *libpfm4* [Eranian and Richter 2018], do not support the hardware performance events that are required to directly measure data traffic throughout the memory hierarchy of AMD Epyc processors. Therefore, we focus on the Intel-based systems for now, although

Table 3. Serial performance (in Gflop/s) of a CSR-based SpMV kernel. The first column is the performance of a reference implementation on Sandy Bridge provided by the function `mkl_cspblas_dcsrgemv` from the Intel Math Kernel Library (MKL) 2017. The remaining columns compare the performance of our implementation compiled with GCC and ICC on both Sandy Bridge and Epyc.

	Sandy Bridge			Epyc	
Matrix	Intel MKL 2017	GCC 7.4.0	ICC 17.0.0	GCC 7.4.0	ICC 19.0.5
TSOPF_RS_b2383	1.60	1.54	1.93	1.67	2.61
spal_004	1.11	1.15	1.19	1.55	2.15
RM07R	1.36	1.51	1.76	1.76	2.45
relat9	0.64	0.67	0.59	1.04	0.92
HV15R	1.38	1.34	1.66	1.77	2.49
GL7d19	0.26	0.26	0.26	0.39	0.36
sx-stackoverflow	0.23	0.25	0.23	0.35	0.32
FullChip	0.90	0.97	0.92	1.26	1.19
Freescale1	0.87	0.97	0.86	1.37	1.27
circuit5M	1.14	1.23	1.17	1.74	1.88
Hardesty3	1.02	1.14	0.98	1.74	1.60
Lynx68	0.21	0.21	0.21	0.34	0.30
Lynx68_reordered	0.66	0.65	0.69	1.33	1.24

we consider the AMD Epyc system once again when evaluating the performance of the CSR-based SpMV kernel in Section 5.5. Measurements of the actual data traffic volumes are obtained from Performance Monitoring Units (PMUs) in the hardware that can be configured to report various hardware performance events [Intel Corporation 2017, Ch. 18]. In our case, these events are accessed by providing appropriate event encodings to the `perf_event_open` system call, and using the returned file descriptor to read values from hardware performance counters. The library *libpfm4* [Eranian and Richter 2018] is used to translate event names to event encodings. Moreover, we follow guidelines given by Molka et al. [2017] to select appropriate hardware performance events for data traffic measurements by using microbenchmarks to correlate hardware events with data transfers between levels of the memory hierarchy. The events that have been chosen as a result of this procedure are shown in Table 4.

For each matrix in Table 1, we compute SpMV in parallel using Algorithm 2, and, for each thread, we record the hardware performance events listed in Table 4. These events provide the number of cache misses per core for each cache level, from which the total data traffic volumes are computed.

The estimated and measured total data traffic volumes for CSR SpMV using a single core, single socket, and both sockets are shown in Tables 5 and 6 for Sandy Bridge and Skylake, respectively. Also, estimated data traffic volumes for the single core and single socket cases on AMD Epyc are shown in Table 7. In the following, we refer to the data traffic volume from the L2 cache to the L1 cache as L1 $\leftarrow$ L2 traffic, and so on for the remaining levels of the hierarchy.

Overall, the correspondence between measured and estimated data traffic is close,

Table 4. Hardware performance events recorded for each processor core. Each event includes cache lines brought to the specified cache due to loads, stores, and hardware prefetches. Event names follow the conventions of the library libpfm4 [Eranian and Richter 2018].

Cache level	Sandy Bridge	Skylake
L1 ← L2	`l1-dcache-load-misses`	`l1-dcache-load-misses`
L2 ← L3	`l2_lines_in:any`	`l2_lines_in:any`
L3 ← DRAM	`offcore_response_0: any_data:any_rfo: l3_miss:snp_any`	`offcore_response_0: any_data:any_rfo: l3_miss_local: l3_miss_miss_remote_hop1_dram: snp_any`

which shows that the main contributions to data traffic are captured by our simulation scheme, even with a simplified model of the memory hierarchy. Most estimates are within a few percent of the measured data traffic volumes, but there are some notable exceptions.

First, for some of the most irregular matrices, `spal_004`, `GL7d19`, `Lynx68`, and `sx-stackoverflow`, our method underestimates the L1 ← L2 traffic on both Sandy Bridge and Skylake by about 15–40 %. Also, the L2 ← L3 traffic on Sandy Bridge is underestimated by about 10–20 % for `spal_004` and `Lynx68`. These discrepancies are likely caused by conflict misses due to the caches being set-associative, rather than fully associative, as we assume in our simplified model of the memory hierarchy. Observe how the estimates of the L2 ← L3 traffic on Skylake are more accurate, probably due to the increased cache size and associativity, which should result in fewer conflict misses. Now, there is also another, plausible reason for underestimating the data traffic volume: There may be additional, needless data traffic in cases where hardware prefetching mechanisms place data into a CPU cache and the data is later evicted without being used. This is possible because hardware prefetchers may transfer data from lower levels of a memory hierarchy, even though there are no outstanding requests referencing the data. The indirect and irregular memory access patterns that occur in SpMV can easily render standard hardware prefetching mechanisms inefficient, though this will naturally depend a great deal on the sparsity pattern of the matrix. While it would be possible to assess the impact of hardware prefetchers by explicitly disabling them, this is beyond the scope of our current work.

Second, the cache simulation overestimates the L3 ← DRAM for the matrices `sx-stackoverflow` and `Lynx68` on Skylake by about ten and forty percent, respectively. The fact that this does not occur on Sandy Bridge suggests that the overestimation is related to the L3 cache not being inclusive of the L2 cache on Skylake. It is likely that a more accurate estimate could be produced if non-inclusivity were taken into account, for example, by combining the size of the L2 and L3 caches.

Table 5. Estimated and measured total data traffic (in MiB) on Intel Sandy Bridge Xeon E5-2650 for SpMV with Algorithm 2.

	L1 ← L2			L2 ← L3			L3 ← DRAM		
Matrix	Meas. [MiB]	Est. [MiB]	Err. [%]	Meas. [MiB]	Est. [MiB]	Err. [%]	Meas. [MiB]	Est. [MiB]	Err. [%]
Single core									
TSOPF_RS_b2383	188	186	-1.1	187	186	-0.5	186	186	0.0
spal_004	1335	1054	-21.0	956	839	-12.2	529	528	-0.2
RM07R	503	488	-3.0	472	456	-3.4	441	439	-0.5
relat9	1505	1477	-1.9	1114	1086	-2.5	594	595	0.2
HV15R	3597	3522	-2.1	3480	3397	-2.4	3308	3291	-0.5
GL7d19	3526	2714	-23.0	2679	2631	-1.8	573	532	-7.2
sx-stackoverflow	3429	2565	-25.2	2600	2446	-5.9	749	717	-4.3
FullChip	500	490	-2.0	474	467	-1.5	454	447	-1.5
Freescale1	387	381	-1.6	385	375	-2.6	347	344	-0.9
circuit5M	1180	1185	0.4	923	958	3.8	937	944	0.7
Hardesty3	742	734	-1.1	650	618	-4.9	616	618	0.3
Lynx68	9103	5560	-38.9	6873	5532	-19.5	4367	4270	-2.2
Lynx68_reordered	5297	5131	-3.1	2965	2891	-2.5	1428	1428	0.0
Single socket									
TSOPF_RS_b2383	188	186	-1.1	188	186	-1.1	185	186	0.5
spal_004	1344	1054	-21.6	962	840	-12.7	515	529	2.7
RM07R	504	488	-3.2	474	456	-3.8	420	438	4.3
relat9	1506	1477	-1.9	1112	1086	-2.3	584	591	1.2
HV15R	3600	3522	-2.2	3493	3397	-2.7	3220	3357	4.3
GL7d19	3527	2714	-23.1	2679	2631	-1.8	751	673	-10.4
sx-stackoverflow	3433	2565	-25.3	2592	2615	-6.5	791	749	-5.5
FullChip	501	490	-2.2	481	467	-2.9	427	439	2.8
Freescale1	388	381	-1.8	389	375	-3.6	327	338	3.4
circuit5M	1181	1185	0.3	928	958	3.2	905	943	4.2
Hardesty3	742	734	-1.1	658	619	-5.9	576	618	7.3
Lynx68	9105	5560	-38.9	6923	5532	-20.1	4418	4316	-2.3
Lynx68_reordered	5302	5131	-3.2	2996	2891	-3.5	1467	1465	-0.1
Dual socket									
TSOPF_RS_b2383	188	186	-1.1	187	185	-1.1	185	185	0.0
spal_004	1346	1054	-21.7	959	840	-12.4	542	531	-2.0
RM07R	504	488	-3.2	475	456	-4.0	451	439	-2.7
relat9	1510	1477	-2.2	1113	1087	-2.3	591	592	0.2
HV15R	3602	3522	-2.2	3499	3397	-2.9	3413	3357	-1.6
GL7d19	3528	2714	-23.1	2688	2631	-2.1	617	563	-8.8
sx-stackoverflow	3434	2565	-25.3	2613	2446	-6.4	795	732	-7.9
FullChip	502	490	-2.4	483	467	-3.3	449	438	-2.4
Freescale1	388	381	-1.8	390	375	-3.8	354	348	-1.7
circuit5M	1181	1185	0.3	928	958	3.2	895	917	2.5
Hardesty3	742	734	-1.1	668	619	-7.3	620	618	-0.3
Lynx68	9099	5560	-38.9	6937	5532	-20.3	4402	4290	-2.5
Lynx68_reordered	5302	5131	-3.2	3054	2892	-5.3	1481	1463	-1.2

Table 6. Estimated and measured total data traffic (in MiB) on Intel Skylake Xeon Platinum 8168 for SpMV with Algorithm 2.

	L1 ← L2			L2 ← L3			L3 ← DRAM		
Matrix	Meas. [MiB]	Est. [MiB]	Err. [%]	Meas. [MiB]	Est. [MiB]	Err. [%]	Meas. [MiB]	Est. [MiB]	Err. [%]
Single core									
TSOPF_RS_b2383	188	186	-1.1	187	186	-0.5	187	186	-0.5
spal_004	1332	1054	-20.9	858	838	-2.3	554	528	-4.7
RM07R	503	488	-3.0	471	454	-3.6	456	439	-3.7
relat9	1497	1477	-1.3	878	842	-4.1	595	591	-0.7
HV15R	3598	3522	-2.1	3446	3368	-2.3	3383	3287	-2.8
GL7d19	3209	2714	-15.4	2401	2397	-0.2	504	479	-5.0
sx-stackoverflow	3240	2565	-20.8	2190	2173	-0.8	519	552	6.4
FullChip	500	490	-2.0	482	461	-4.4	431	436	1.2
Freescale1	387	381	-1.6	388	369	-4.9	339	340	0.3
circuit5M	1180	1185	0.4	963	957	-0.6	933	915	-1.9
Hardesty3	741	734	-0.9	623	618	-0.8	622	618	-0.6
Lynx68	8729	5560	-36.3	5551	5480	-1.3	3260	3539	8.6
Lynx68_reordered	5247	5131	-2.2	1584	1529	-3.5	1497	1424	-4.9
Single socket									
TSOPF_RS_b2383	191	186	-2.6	192	185	-3.6	186	186	0.0
spal_004	1318	1054	-20.0	860	838	-2.6	532	528	-0.8
RM07R	518	488	-5.8	474	454	-4.2	442	448	1.4
relat9	1500	1477	-1.5	871	843	-3.2	591	588	-0.5
HV15R	3625	3522	-2.8	3456	3368	-2.5	3382	3360	-0.7
GL7d19	3210	2714	-15.5	2420	2397	-1.0	484	456	-5.8
sx-stackoverflow	3243	2565	-20.9	2263	2173	-4.0	517	555	7.4
FullChip	504	490	-2.8	513	461	-10.1	430	428	-0.5
Freescale1	388	381	-1.8	395	369	-6.6	333	332	-0.3
circuit5M	1189	1185	-0.3	1058	957	-9.5	935	930	-0.5
Hardesty3	742	734	-1.1	634	619	-2.4	625	618	-1.1
Lynx68	8732	5560	-36.3	5594	5480	-2.0	2589	3599	39.0
Lynx68_reordered	5249	5131	-2.2	1635	1535	-6.1	1509	1494	-1.0
Dual socket									
TSOPF_RS_b2383	194	187	-3.6	199	185	-7.0	187	185	-1.1
spal_004	1320	1054	-20.2	866	840	-3.0	535	528	-1.3
RM07R	525	488	-7.0	481	454	-5.6	441	447	1.4
relat9	1504	1477	-1.8	882	844	-4.3	593	589	-0.7
HV15R	3683	3522	-4.4	3467	3368	-2.9	3346	3360	0.4
GL7d19	3213	2714	-15.5	2425	2397	-1.2	483	473	-2.1
sx-stackoverflow	3254	2565	-21.2	2342	2173	-7.2	504	552	9.5
FullChip	511	490	-4.1	544	462	-15.1	410	428	4.4
Freescale1	389	381	-2.1	398	369	-7.3	316	332	5.1
circuit5M	1196	1185	-0.9	1073	957	-10.8	908	927	2.1
Hardesty3	744	734	-1.3	653	620	-5.1	632	618	-2.2
Lynx68	8734	5560	-36.3	5623	5480	-2.5	2586	3564	37.8
Lynx68_reordered	5254	5131	-2.3	1658	1540	-7.1	1488	1482	-0.4

Table 7. Estimated total data traffic (in MiB) on AMD Epyc 7601 for SpMV with Algorithm 2. Measurements of the data traffic on Epyc are missing due to a lack of reliable hardware performance events for measuring data traffic throughout the memory hierarchy.

Matrix	L1 ← L2 [MiB]	L2 ← L3 [MiB]	L3 ← DRAM [MiB]
Single core			
TSOPF_RS_b2383	187	186	186
spal_004	1054	839	530
RM07R	488	455	439
relat9	1477	967	618
HV15R	3522	3369	3301
GL7d19	2714	2548	1111
sx-stackoverflow	2565	2340	1207
FullChip	490	463	457
Freescale1	381	371	351
circuit5M	1185	958	945
Hardesty3	734	618	618
Lynx68	5560	5514	5017
Lynx68_reordered	5131	1786	1438
Single socket			
TSOPF_RS_b2383	186	185	185
spal_004	1054	840	607
RM07R	488	456	446
relat9	1477	968	614
HV15R	3522	3369	3359
GL7d19	2714	2548	1136
sx-stackoverflow	2565	2340	1219
FullChip	490	463	453
Freescale1	381	371	359
circuit5M	1185	958	937
Hardesty3	734	619	618
Lynx68	5560	5514	5020
Lynx68_reordered	5131	1792	1475

5.4 Measuring CPU cache and memory bandwidth

Before we can relate the relevant data traffic volumes in a memory hierarchy to an algorithm's performance, we must first benchmark the CPU cache and main memory bandwidths of our test systems. We do so by using a modified version of the STREAM bandwidth benchmark [McCalpin 1995].

The original STREAM benchmark only includes computational kernels with regular memory access patterns, and it is not clear that the measured bandwidths are relevant for an irregular computation, such as SpMV. For example, the standard kernels benefit greatly from the compiler's automatic vectorisation, but the CSR SpMV kernel in Algorithm 2 usually does not. In addition, the STREAM kernels do not include indirect memory accesses, and, moreover, they perform a larger proportion of writes than

Table 8. Memory and CPU cache bandwidth (in GB/s) measured by the STREAM benchmark [McCalpin 1995]. In addition to the standard *Triad* kernel, we have added a kernel for computing a dot product with indirect memory accesses. The latter kernel is representative of the SpMV computation in Algorithm 2, and it corresponds to SpMV with a matrix in the CSR format that has a single, dense row. The bandwidths per NUMA domain and per socket are measured by using *numactl* to place memory in a single NUMA domain or socket, and running the benchmarks with all cores belonging to the given NUMA domain or socket, respectively.

	Triad ($c_k \leftarrow a_k + b_k \cdot d$)			Indirect dot product ($c \leftarrow \sum_k a_k \cdot b_{j_k},\ j_k = k$)		
	Sandy Bridge	Skylake	Epyc	Sandy Bridge	Skylake	Epyc
Registers ↔ L1, per core	81.4	212.0	63.2	13.1	13.2	8.7
L1 ↔ L2, per core	29.2	88.8	64.9	13.3	13.4	8.8
L2 ↔ L3, per core	17.9	24.2	51.3	12.7	12.5	8.8
L3 ↔ DRAM						
per core	9.1	11.4	18.3	9.8	10.1	8.7
per NUMA domain	28.1	74.8	21.0	37.3	99.8	29.6
per socket	28.1	74.8	85.4	37.3	99.8	118.5
all sockets	56.8	139.2	168.4	70.6	184.5	225.7

is typical for SpMV. Therefore, we supplement the STREAM benchmark with an additional computational kernel, which more closely resembles SpMV, and consists of computing a dot product with indirect memory accesses. This is similar to the approach used by Vuduc et al. [2002].

The measured bandwidths for the different kernels are shown in Table 8. The sizes of the arrays used by each kernel are selected so that the data fits in the appropriate level of the memory hierarchy, and times are measured using `clock_gettime` with `CLOCK_MONOTONIC`. The Triad kernel is included mostly as a reference, since the numbers it produces are often used in performance evaluations, for example, in the Roofline model [Williams et al. 2009b].

The Triad kernel has a higher throughput than the indirect dot product whenever data is read from one of the CPU caches. This is because the compiler generates vectorised code for Triad, which alleviates bottlenecks that are limiting the performance of the indirect dot product kernel. Meanwhile, for the per-socket memory bandwidth, the Triad kernel produces a result that is about 75 % of the indirect dot product. This difference is caused by the following two facts. First, the indirect dot product performs no stores, whereas the Triad kernel performs one store for every two loads. Second, the bandwidths reported by STREAM assume that the traffic generated by a store is the same as for a load. However, in practice, stores usually generate twice as much data traffic, that is, sixteen bytes per store for Triad, because a cache line is transferred from main memory before a store is performed, and, in addition, the modified cache line is written back to main memory afterwards. Although it is possible to use non-temporal stores to avoid bringing cache lines from main memory before a store, this requires either using a different compiler, such as the Intel C Compiler, or hand-coded

optimisation, because GCC will not generate such code.

5.5 Evaluating performance for CSR-based SpMV

In this section, we apply the performance model from Section 3 to evaluate the performance of the standard CSR SpMV algorithm. The performance model uses the data traffic estimates from Section 5.3 and CPU cache and memory bandwidths from Section 5.4.

First, we record the actual execution time and performance, so that we can later compare to our performance predictions. For each matrix in Table 1, we compute SpMV in parallel using Algorithm 2 and measure the execution time. To account for variations in the measurements, we obtain a random sample of one hundred trials, and let T denote the sample mean execution time. The performance is $S = 2N/T$, where N is the number of matrix non-zeros.

For each cache level, we compute upper bounds on performance based on Eq. (1) using per-core data traffic and bandwidth. That is, the maximum of the data traffic volumes among all cores is used along with the single-core bandwidth of data transfers to the corresponding level of the memory hierarchy. Finally, there is an upper bound on performance based on Eq. (2) and the total traffic and bandwidth of each NUMA domain. For the bandwidths, we have used the numbers given in Table 8 for the "indirect dot product" kernel.

Upper bounds on performance for CSR-based SpMV on Sandy Bridge, Skylake, and Epyc are shown in Tables 9, 10 and 11. Again, for the Intel systems, we present results using a single core, single socket and both sockets. For AMD Epyc, we present single core and single socket results.

Generally speaking, no single level of the memory hierarchy can be regarded as a bottleneck for all of the tested matrices. For a single core, bottlenecks sometimes appear between the registers and L1 cache, between L1 and L2, and also between L3 and main memory. Whenever additional cores are used, performance is frequently limited by the aggregate memory bandwidth. However, for some matrices, including `GL7d19`, `sx-stackoverflow`, and `Lynx68_reordered`, L1 $\leftarrow$ L2 traffic remains the limiting factor. In these cases, the aggregate memory bandwidth per socket or NUMA domain is plentiful, but the irregular traffic between the L1 and L2 cache is so substantial that it limits performance. In addition, the sparsity patterns of `circuit5m` and `FullChip` result in severely unbalanced workloads among multiple processors, which is not surprising due to both matrices having at least one very long row. Note that the performance bounds increase only marginally when moving from a single core to multiple cores, far from the ideal speedup one might hope for with a perfectly balanced workload. Moreover, in both cases, the per-core bounds are lower than the per-socket or per NUMA domain bounds, indicating that the aggregate memory bandwidth is not saturated.

Comparing the two Intel systems, our predictions show very little difference between Sandy Bridge and Skylake in the single-core case. However, when looking at an entire socket or full machine, the Skylake system is predicted to perform much better, mostly due to its higher aggregate memory bandwidth. In addition, the increased size of Skylake's L2 cache means that some matrices, such as `Lynx68_reordered`, generate

Table 9. Upper bounds on performance (in Gflop/s) on Intel Sandy Bridge Xeon E5-2650 for SpMV with Algorithm 2. The column "Reg. ← L1" is based on the number of loads from the L1 cache to registers. The next four columns are based on the estimated data traffic volumes in Table 5. The "per core" bounds use the maximum data traffic volume among all cores together with the single-core CPU cache or memory bandwidth. The "per socket" bound is based on the largest of the per-socket traffic volumes and the aggregate memory bandwidth of a single socket. The relevant bandwidths are from the "indirect dot product" kernel in Table 8. The smallest of the upper bounds is displayed in bold and represents the bottleneck predicted by our model.

Matrix	Reg. ← L1 per core	L1 ← L2 per core	L2 ← L3 per core	L3 ← DRAM per core	L3 ← DRAM per socket
Single core					
TSOPF_RS_b2383	**1.31**	2.21	2.11	1.63	—
spal_004	1.31	**1.11**	1.33	1.63	—
RM07R	**1.31**	1.95	1.99	1.60	—
relat9	1.31	**0.67**	0.87	1.22	—
HV15R	**1.31**	2.04	2.02	1.61	—
GL7d19	1.31	0.35	**0.34**	1.31	—
sx-stackoverflow	1.31	**0.36**	0.36	0.94	—
FullChip	1.31	1.38	1.38	**1.11**	—
Freescale1	1.31	1.26	1.22	**1.03**	—
circuit5M	1.31	1.27	1.51	**1.18**	—
Hardesty3	1.31	1.40	1.59	**1.22**	—
Lynx68	1.31	0.51	0.49	**0.49**	—
Lynx68_reordered	1.31	**0.55**	0.93	1.46	—
Single socket					
TSOPF_RS_b2383	**4.52**	7.60	7.25	5.60	6.19
spal_004	10.38	8.25	10.55	12.88	**6.21**
RM07R	9.92	14.62	15.13	12.07	**6.09**
relat9	8.26	**3.92**	5.10	8.18	4.69
HV15R	10.11	15.61	15.51	12.11	**6.00**
GL7d19	9.91	**2.65**	2.67	6.91	3.95
sx-stackoverflow	3.30	**0.90**	0.92	2.61	3.44
FullChip	4.21	3.46	3.49	**3.00**	4.31
Freescale1	5.99	5.11	4.98	4.53	**3.98**
circuit5M	2.21	**1.86**	2.47	1.95	4.49
Hardesty3	10.32	10.92	12.56	9.69	**4.66**
Lynx68	10.30	3.40	3.25	3.27	**1.84**
Lynx68_reordered	10.40	**4.35**	7.34	11.27	5.42
Dual socket					
TSOPF_RS_b2383	9.04	15.19	14.51	11.20	**6.22**
spal_004	24.42	16.50	21.10	25.38	**12.35**
RM07R	19.82	29.70	30.25	24.15	**11.79**
relat9	16.52	**7.66**	9.93	16.18	7.74
HV15R	20.21	31.22	31.03	24.27	**11.98**
GL7d19	18.78	**5.06**	5.20	19.38	9.22
sx-stackoverflow	4.80	**1.31**	1.36	4.67	5.36
FullChip	5.21	**4.19**	4.19	4.48	7.43
Freescale1	10.83	8.73	8.49	9.31	**6.70**
circuit5M	**3.36**	3.66	3.62	3.55	6.65
Hardesty3	20.64	21.83	25.12	19.39	**9.22**
Lynx68	20.60	6.79	6.51	9.35	**3.48**
Lynx68_reordered	20.73	**8.65**	14.53	22.42	10.75

Table 10. Upper bounds on performance (in Gflop/s) on Intel Skylake Xeon Platinum 8168 for SpMV with Algorithm 2. The column "Reg. ← L1" is based on the number of loads from the L1 cache to registers. The next four columns are based on the estimated data traffic volumes in Table 6. The "per core" bounds use the maximum data traffic volume among all cores together with the single-core CPU cache or memory bandwidth. The "per socket" bound is based on the largest of the per-socket traffic volumes and the aggregate memory bandwidth of a single socket. The relevant bandwidths are from the "indirect dot product" kernel in Table 8. The smallest of the upper bounds is displayed in bold and represents the bottleneck predicted by our model.

Matrix	Reg. ← L1 per core	L1 ← L2 per core	L2 ← L3 per core	L3 ← DRAM per core	L3 ← DRAM per socket
Single core					
TSOPF_RS_b2383	**1.32**	2.22	2.07	1.67	—
spal_004	1.32	**1.12**	1.31	1.68	—
RM07R	**1.32**	1.96	1.97	1.64	—
relat9	1.32	**0.67**	1.10	1.27	—
HV15R	**1.32**	2.05	2.00	1.66	—
GL7d19	1.32	**0.35**	0.37	1.50	—
sx-stackoverflow	1.32	**0.36**	0.40	1.26	—
FullChip	1.32	1.39	1.38	**1.18**	—
Freescale1	1.32	1.27	1.22	**1.07**	—
circuit5M	1.32	1.28	1.48	**1.25**	—
Hardesty3	1.32	1.41	1.56	**1.26**	—
Lynx68	1.32	0.51	**0.49**	0.61	—
Lynx68_reordered	1.32	**0.56**	1.74	1.51	—
Single socket					
TSOPF_RS_b2383	**13.67**	22.96	21.42	17.31	16.55
spal_004	30.70	24.58	30.58	38.67	**16.64**
RM07R	29.46	43.52	44.66	36.09	**15.92**
relat9	24.96	**11.71**	19.76	25.88	12.61
HV15R	28.38	43.58	42.72	34.73	**16.04**
GL7d19	26.92	**7.28**	8.39	31.26	15.58
sx-stackoverflow	5.80	**1.59**	1.96	6.02	12.43
FullChip	5.81	4.69	4.53	**4.35**	11.84
Freescale1	15.99	12.40	12.53	13.50	**10.85**
circuit5M	3.02	**2.27**	3.22	2.76	12.18
Hardesty3	31.19	33.35	37.09	29.97	**12.46**
Lynx68	31.10	10.26	9.71	12.35	**5.90**
Lynx68_reordered	31.29	**12.96**	40.30	33.58	14.21
Dual socket					
TSOPF_RS_b2383	27.31	45.92	42.84	34.61	**16.64**
spal_004	59.30	47.20	57.93	74.12	**33.29**
RM07R	53.36	79.80	81.20	65.61	**31.01**
relat9	49.93	22.13	38.70	50.03	**20.89**
HV15R	56.73	87.17	85.43	69.91	**31.96**
GL7d19	46.53	**12.72**	16.79	55.31	29.85
sx-stackoverflow	8.16	**2.24**	2.85	9.56	17.73
FullChip	6.42	5.23	**4.96**	5.83	20.35
Freescale1	30.86	21.03	21.48	28.04	**18.86**
circuit5M	5.17	**3.85**	5.20	5.26	17.59
Hardesty3	62.39	64.62	74.19	59.94	**24.68**
Lynx68	62.19	20.51	19.41	35.81	**11.48**
Lynx68_reordered	62.27	**25.68**	80.59	67.15	28.43

Table 11. Upper bounds on performance (in Gflop/s) on AMD Epyc 7601 for SpMV with Algorithm 2. The column "Reg. ← L1" is based on the number of loads from the L1 cache to registers. The next four columns are based on the estimated data traffic volumes in Table 7. The "per core" bounds use the maximum data traffic volume among all cores together with the single-core CPU cache or memory bandwidth. The "per NUMA domain" bound is based on the maximum traffic volume among NUMA domains and the aggregate bandwidth of a single NUMA domain. The relevant bandwidths are from the "indirect dot product" kernel in Table 8. The smallest of the upper bounds is displayed in bold and represents the bottleneck predicted by our model.

Matrix	Reg. ← L1 per core	L1 ← L2 per core	L2 ← L3 per core	L3 ← DRAM per core	L3 ← DRAM per socket
Single core					
TSOPF_RS_b2383	**0.88**	1.46	1.46	1.44	—
spal_004	0.88	**0.74**	0.92	1.45	—
RM07R	**0.88**	1.29	1.38	1.42	—
relat9	0.88	**0.44**	0.68	1.05	—
HV15R	**0.88**	1.35	1.41	1.42	—
GL7d19	0.88	**0.23**	0.25	0.56	—
sx-stackoverflow	0.88	**0.24**	0.26	0.50	—
FullChip	**0.88**	0.91	0.97	0.97	—
Freescale1	0.88	**0.83**	0.86	0.89	—
circuit5M	0.88	**0.84**	1.04	1.05	—
Hardesty3	**0.88**	0.93	1.10	1.09	—
Lynx68	0.88	**0.34**	0.34	0.37	—
Lynx68_reordered	0.88	**0.36**	1.05	1.29	—
Single socket					
TSOPF_RS_b2383	12.14	20.88	20.88	10.32	**8.54**
spal_004	27.83	21.53	29.80	20.16	**17.16**
RM07R	25.20	36.99	39.30	20.72	**18.25**
relat9	22.19	**10.06**	15.57	14.05	11.52
HV15R	25.12	38.01	39.93	20.88	**18.36**
GL7d19	22.74	**6.08**	7.37	12.39	6.24
sx-stackoverflow	4.43	**1.20**	1.41	3.78	4.31
FullChip	4.04	**3.19**	3.29	4.70	9.34
Freescale1	14.18	10.24	10.95	10.13	**8.49**
circuit5M	3.04	**2.50**	3.28	4.20	6.67
Hardesty3	27.73	28.29	33.95	16.78	**14.56**
Lynx68	27.64	8.96	9.04	10.06	**4.47**
Lynx68_reordered	27.75	**11.28**	32.28	19.90	17.94

less L2 ← L3 data traffic compared to Sandy Bridge. However, this does not seem to significantly impact performance because data transfers between the L2 and L3 caches never constituted a bottleneck to begin with.

For Epyc, the single core performance is constrained by traffic between registers and the L1 cache or the L1 and L2 caches. This stems from the fact that the measured per core bandwidths of the indirect dot product kernel are essentially the same regardless of which level of the memory hierarchy that data is read from, and there is naturally more traffic to the smaller, higher-level caches. For a single socket, the performance predictions are quite similar to a single socket on the Skylake system. The bottlenecks associated with each matrix are at the same level of the memory hierarchy, with the only exception being `TSOPF_RS_b2383`, which is foremost limited by the aggregate L3 ← DRAM traffic on Epyc and Reg. ← L1 traffic on Sandy Bridge and Skylake.

Finally, our performance predictions are compared to the measured performance in Table 12 for Sandy Bridge and Skylake, and in Table 13 for Epyc. For comparison, we also compute a "best case" upper bound using the best case estimate of the data traffic from Eq. (5) and the main memory bandwidth for the number of cores being used.

For matrices such as `RM07R`, `HV15R`, and `Hardesty3`, whose sparsity patterns are regular and lead to good load balancing, the best case estimates are very close to the measured performance, but our cache simulation estimates are at least of the same quality. On the other hand, the two methods produce significantly different results for matrices with irregular sparsity patterns that cause a large amount of additional, irregular data traffic. Examples of such irregular matrices are `circuit5m`, `GL7d19`, `sx-stackoverflow`, `FullChip`, `Freescale1`, and `Lynx68`. The performance predicted by the cache simulation is always within a factor of three of the measured performance. Compare this to the prediction based on the best case data traffic, which is off by a factor of three or more in 23 out of 72 cases on Intel and 8 out of 36 cases on Epyc. Moreover, among the most irregular matrices, for example, `sx-stackoverflow`, the best case prediction misses by more than a factor of ten.

In cases such as `GL7d19`, `sx-stackoverflow`, and `Lynx68`, where there are still discrepancies between the measured and performance predicted by the cache simulation on Sandy Bridge and Skylake, it is likely that the irregular sparsity patterns of these matrices result in the SpMV computation being limited more by memory latency than bandwidth. Regarding the single core estimates on Epyc, they appear to consistently underestimate the performance, suggesting that the per core bandwidths obtained from the indirect dot product kernel is actually lower than the real throughput that is achieved during CSR SpMV.

5.6 Estimating data traffic and performance for COO-based SpMV

The measured and estimated total data traffic volumes for COO SpMV with Algorithm 3 are shown in Table 14. To keep the discussion short, we show results for the case of a single core and a single socket on Sandy Bridge. For a single core, the results are quite similar to those presented in Table 5 for CSR SpMV in terms of accuracy. The L1 ← L2 and L2 ← L3 traffic is underestimated for a few matrices, likely due to conflict misses. In the single socket case, the data traffic for the private L1 and L2

Table 12. Comparison of measured and estimated performance (in Gflop/s) for CSR-based SpMV with Algorithm 2 on Sandy Bridge and Skylake. The "best case" predictions are based on the best case data traffic estimate Eq. (5) and main memory bandwidth. The "cache sim." predictions produced by our method are the smallest of the upper bounds from Tables 9 and 10. In many cases, our method quantifies performance bottlenecks more accurately and attributes them to data transfers to and from various levels in the memory hierarchy.

	Sandy Bridge					Skylake				
		Best case		Cache sim.			Best case		Cache sim.	
Matrix	Meas. [Gflop/s]	Est. [Gflop/s]	Err. [%]	Est. [Gflop/s]	Err. [%]	Meas. [Gflop/s]	Est. [Gflop/s]	Err. [%]	Est. [Gflop/s]	Err. [%]
Single core										
TSOPF_RS_b2383	1.14	1.63	43	1.31	15	1.19	1.68	41	1.32	11
spal_004	0.85	1.63	92	1.11	31	0.88	1.68	91	1.12	28
RM07R	1.10	1.61	46	1.31	19	1.20	1.66	38	1.32	10
relat9	0.51	1.23	142	0.67	32	0.66	1.27	91	0.67	1
HV15R	1.12	1.61	44	1.31	17	1.20	1.66	39	1.32	10
GL7d19	0.19	1.50	691	0.34	81	0.36	1.55	334	0.35	-2
sx-stackoverflow	0.17	1.46	737	0.36	106	0.35	1.50	331	0.36	4
FullChip	0.76	1.38	81	1.11	46	1.05	1.42	35	1.18	12
Freescale1	0.74	1.25	69	1.03	39	1.09	1.29	19	1.07	-2
circuit5M	0.93	1.61	73	1.18	27	1.12	1.66	48	1.25	12
Hardesty3	0.86	1.23	44	1.22	43	1.18	1.27	7	1.26	7
Lynx68	0.16	1.48	821	0.49	203	0.23	1.53	570	0.49	103
Lynx68_reordered	0.59	1.48	149	0.55	-7	0.80	1.53	91	0.56	-31
Single socket										
TSOPF_RS_b2383	3.77	6.19	64	4.52	20	11.40	16.57	45	13.67	20
spal_004	4.87	6.19	27	6.21	28	14.21	16.55	17	16.64	17
RM07R	5.83	6.11	5	6.09	4	16.71	16.36	-2	15.92	-5
relat9	2.60	4.69	80	3.92	51	8.97	12.54	40	11.71	31
HV15R	5.09	6.14	21	6.00	18	15.07	16.44	9	16.04	6
GL7d19	1.29	5.72	343	2.65	105	7.52	15.31	104	7.28	-3
sx-stackoverflow	.51	5.55	998	.90	78	1.85	14.86	702	1.59	-14
FullChip	1.88	5.24	178	3.00	59	3.51	14.01	299	4.35	24
Freescale1	2.49	4.77	91	3.98	60	8.26	12.77	55	10.85	31
circuit5M	1.47	6.12	317	1.86	27	2.20	16.38	646	2.27	3
Hardesty3	3.58	4.68	31	4.66	30	9.09	12.52	38	12.46	37
Lynx68	1.03	5.64	449	1.84	79	4.96	15.10	204	5.90	19
Lynx68_reordered	3.38	5.64	67	4.35	29	11.65	15.10	30	12.96	11
Dual socket										
TSOPF_RS_b2383	5.57	11.72	110	6.22	12	13.36	30.63	129	16.64	25
spal_004	9.00	11.71	30	12.35	37	26.58	30.60	15	33.29	25
RM07R	10.61	11.57	9	11.79	11	31.81	30.24	-5	31.01	-3
relat9	4.74	8.87	87	7.66	62	15.44	23.18	50	20.89	35
HV15R	11.09	11.63	5	11.98	8	29.11	30.39	4	31.96	10
GL7d19	2.52	10.83	329	5.06	101	13.46	28.31	110	12.72	-6
sx-stackoverflow	.76	10.51	1287	1.31	73	2.70	27.46	916	2.24	-17
FullChip	2.43	9.91	309	4.19	73	4.71	25.91	450	4.96	5
Freescale1	4.21	9.04	114	6.70	59	15.81	23.62	49	18.86	19
circuit5M	2.32	11.59	400	3.36	45	3.87	30.28	683	3.85	-0
Hardesty3	5.98	8.86	48	9.22	54	16.26	23.15	42	24.68	52
Lynx68	1.98	10.68	438	3.48	75	9.31	27.91	200	11.48	23
Lynx68_reordered	6.92	10.68	54	8.65	25	23.33	27.91	20	25.68	10

Table 13. Comparison of measured and estimated performance (in Gflop/s) for CSR-based SpMV with Algorithm 2 on Epyc. The "best case" predictions are based on the best case data traffic estimate Eq. (5) and main memory bandwidth. The "cache sim." predictions produced by our method are the smallest of the upper bounds from Table 11.

	Epyc, single core					Epyc, single socket				
		Best case		Cache sim.			Best case		Cache sim.	
Matrix	Meas. [Gflop/s]	Est. [Gflop/s]	Err. [%]	Est. [Gflop/s]	Err. [%]	Meas. [Gflop/s]	Est. [Gflop/s]	Err. [%]	Est. [Gflop/s]	Err. [%]
TSOPF_RS_b2383	.89	1.44	62	.88	-2	7.67	19.67	156	8.54	11
spal_004	.84	1.44	72	.74	-12	14.23	19.65	38	17.16	21
RM07R	1.00	1.43	43	.88	-12	17.41	19.42	12	18.25	5
relat9	.88	1.09	24	.44	-50	8.75	14.89	70	10.06	15
HV15R	1.00	1.43	43	.88	-12	15.95	19.52	22	18.36	15
GL7d19	.36	1.34	276	.23	-35	5.15	18.18	253	6.08	18
sx-stackoverflow	.33	1.30	292	.24	-28	1.37	17.64	1184	1.20	-13
FullChip	.84	1.22	46	.88	5	2.69	16.64	519	3.19	19
Freescale1	1.12	1.11	-1	.83	-26	5.78	15.17	162	8.49	47
circuit5M	1.12	1.43	27	.84	-25	2.67	19.45	629	2.50	-6
Hardesty3	1.38	1.09	-21	.88	-36	8.58	14.87	73	14.56	70
Lynx68	.28	1.32	366	.34	19	3.71	17.93	384	4.47	21
Lynx68_reordered	1.03	1.32	27	.36	-65	14.52	17.93	23	11.28	-22

caches is often severely underestimated. Notice, in particular, the matrix spal_004, which has very few rows, and is therefore likely to suffer from false sharing. That is, to maintain cache coherency, two different CPU cores writing to the same cache block may cause the cache block to be transferred from a shared level of the memory hierarchy for every write. Since the L3 cache is shared by all the cores on a single socket, there is no false sharing for the L3 ← DRAM traffic, and the estimates are quite accurate. We suspect that in the dual socket case, requirements for cache coherence between the L3 caches of the two sockets will lead to false sharing and similar issues also for the L3 ← DRAM traffic.

Finally, the performance of the COO SpMV kernel on Sandy Bridge is shown in Table 15 together with performance predictions based on best-case estimates of the memory traffic as well as predictions based on the cache simulation. Five of the matrices benefit somewhat from the cache simulation, whereas there is almost no difference between the two predictions for the rest of the matrices. For a single core, the performance predictions from the cache simulation are within fifty percent of the measured performance for all except four of the matrices. However, the performance impact of going from a single core to a full socket is far from ideal due to the cost of using atomic writes. As a result, the performance predictions become less accurate in general. Moreover, for spal_004 and FullChip there is a significant slowdown as performance is crippled by false sharing. Since our performance model does not account for false sharing, the performance estimates are not even close in these cases.

Table 14. Estimated and measured total data traffic (in MiB) on Intel Sandy Bridge Xeon E5-2650 for COO-based SpMV with Algorithm 3.

	L1 ← L2			L2 ← L3			L3 ← DRAM		
Matrix	Meas. [MiB]	Est. [MiB]	Err. [%]	Meas. [MiB]	Est. [MiB]	Err. [%]	Meas. [MiB]	Est. [MiB]	Err. [%]
Single core									
TSOPF_RS_b2383	296	289	-2.4	281	289	2.8	247	248	0.4
spal_004	776	736	-5.2	766	735	-4.0	706	707	0.1
RM07R	660	626	-5.2	621	595	-4.2	582	579	-0.5
relat9	2902	2143	-26.2	2518	2045	-18.8	1357	1300	-4.2
HV15R	4768	4585	-3.8	4604	4469	-2.9	4380	4361	-0.4
GL7d19	3669	2849	-22.3	2843	2773	-2.5	687	686	-0.1
sx-stackoverflow	3646	2657	-27.1	2560	2371	-7.4	772	791	2.5
FullChip	610	583	-4.4	590	558	-5.4	546	540	-1.1
Freescale1	453	440	-2.9	459	434	-5.4	405	404	-0.2
circuit5M	1421	1390	-2.2	1174	1164	-0.9	1140	1150	0.9
Hardesty3	880	861	-2.2	806	851	5.6	734	738	0.5
Lynx68	9453	5960	-37.0	7291	5933	-18.6	4823	4757	-1.4
Lynx68_reordered	5670	5556	-2.0	3590	3464	-3.5	1831	1830	-0.1
Single socket									
TSOPF_RS_b2383	357	289	-19.0	377	289	-23.3	247	247	0.0
spal_004	6640	736	-88.9	15514	735	-95.3	705	707	0.3
RM07R	666	626	-6.0	632	595	-5.9	591	588	-0.5
relat9	2949	2143	-27.3	2567	2045	-20.3	1702	1603	-5.8
HV15R	4789	4585	-4.3	4626	4469	-3.4	4488	4426	-1.4
GL7d19	3834	2849	-25.7	3006	2773	-7.8	934	885	-5.2
sx-stackoverflow	4279	2657	-37.9	3325	2371	-28.7	819	823	0.5
FullChip	1672	583	-65.1	2237	558	-75.1	545	534	-2.0
Freescale1	457	440	-3.7	466	434	-6.9	407	403	1.0
circuit5M	1509	1390	-7.9	1333	1164	-12.7	1140	1157	1.5
Hardesty3	881	861	-2.3	809	851	5.2	734	738	0.5
Lynx68	9474	5960	-37.1	7343	5933	-19.2	4863	4803	-1.2
Lynx68_reordered	5686	5556	-2.3	3602	3464	-3.8	1891	1876	-0.8

Table 15. Comparison of measured and estimated performance (in Gflop/s) for COO-based SpMV with Algorithm 3 on Sandy Bridge. The "best case" predictions are based on a best case estimate of the data traffic, whereas the "cache sim." predictions are based on estimates of memory traffic produced by our cache simulation for each level of the memory hierarchy.

	Sandy Bridge, single core					Sandy Bridge, single socket				
		Best case		Cache sim.			Best case		Cache sim.	
Matrix	Meas. [Gflop/s]	Est. [Gflop/s]	Err. [%]	Est. [Gflop/s]	Err. [%]	Meas. [Gflop/s]	Est. [Gflop/s]	Err. [%]	Est. [Gflop/s]	Err. [%]
TSOPF_RS_b2383	0.86	1.22	42	1.22	42	0.99	4.65	370	4.65	370
spal_004	0.83	1.22	47	1.22	47	0.33	4.65	1309	4.65	1309
RM07R	0.84	1.21	44	1.21	44	1.30	4.62	255	4.53	248
relat9	0.19	1.05	453	0.46	142	0.65	4.00	515	1.73	166
HV15R	0.85	1.22	44	1.21	42	1.31	4.63	253	4.55	247
GL7d19	0.17	1.16	582	0.33	94	0.69	4.43	542	2.57	272
sx-stackoverflow	0.16	1.14	612	0.35	119	0.55	4.35	691	2.73	396
FullChip	0.64	1.10	72	0.92	44	0.10	4.19	4090	3.54	3440
Freescale1	0.62	1.04	68	0.87	40	1.00	3.95	295	3.33	233
circuit5M	0.74	1.12	51	0.97	31	0.99	4.26	330	3.66	270
Hardesty3	0.76	1.02	34	1.02	34	1.31	3.90	198	3.90	198
Lynx68	0.15	1.15	667	0.44	193	0.61	4.39	620	1.65	170
Lynx68_reordered	0.39	1.15	195	0.51	31	0.79	4.39	456	4.05	413

6 Related work

The cache simulation method we have presented builds on analytical cache models [Agarwal et al. 1989] and trace-driven memory simulation [Uhlig and Mudge 1997], both of which are well known methods for studying cache performance. In their survey, Uhlig and Mudge [1997] compare a number of advanced tools for trace-driven memory simulation that cope with various cache configurations, such as associativity and replacement policies. Our approach is to develop a model that is as simple as possible, but accurate enough to diagnose potential performance issues. Our cache simulation method is somewhat related to the model described by Aho et al. [1971] for page replacements in a virtual memory computer, which is similarly based on counting page replacements generated by a sequence of memory references. In addition, we draw heavily on the ideal-cache model [Frigo et al. 2012], though we explicitly incorporate multi-level hierarchies and shared caches. Shared caches have previously been studied based on statistical models that use high-level information such as a thread's average fetch rate [Sandberg et al. 2011].

Regarding performance models, Langguth et al. [2015a] developed a performance model for irregular applications that was used to study a special case of SpMV, where matrices were derived from a finite volume method on an unstructured, tetrahedral mesh. Our current work may be seen as a continuation that provides a general method for estimating data traffic volumes in a memory hierarchy.

In the context of SpMV, Heras et al. [2001] have used a cache simulation technique for conflict misses in direct-mapped and set-associative caches. Also, Yzelman and Bisseling [2009] have used similar methods to evaluate sparse matrix re-ordering strategies for SpMV, though their simulator is limited to a single cache. Goumas et al.

[2009] and Williams et al. [2009a] have performed experimental evaluations of SpMV performance on various multi-core CPU architectures, identifying several potential performance bottlenecks that depend on matrix structure as well as characteristics of the computing hardware. While Goumas et al. produce a set of guidelines for optimising SpMV kernels, Williams et al. rely on automatic tuning to select appropriate optimisations. In both cases, memory bandwidth is identified as a significant performance bottleneck, but neither of these studies attempt to quantify the impact of data traffic from main memory or any other level of the memory hierarchy. In other work, Vuduc et al. [2002] have modelled SpMV performance based on cache miss estimates, sustainable memory bandwidth and memory access latencies. However, the data traffic volume is based on a best case scenario that is often too optimistic, especially for matrices with highly irregular sparsity patterns. Malossi et al. [2014] proposed to use machine learning techniques to characterise SpMV performance, though the disadvantages are a potentially expensive re-training step needed to calibrate the model for each new hardware architecture, and also that the method does not identify performance bottlenecks explicitly, even though it correlates sparse matrix features with SpMV performance.

7 Conclusion

The performance of irregular, bandwidth-limited computations, such as SpMV, is dictated by data transfers between levels of a CPU's memory hierarchy. Even though it is fairly easy to acquire worst- and best case estimates of the data traffic, these estimates are not always sufficient for locating and quantifying bottlenecks because a precise characterisation of irregular data traffic is missing. We have presented a cache simulation method that accurately quantifies data traffic in a multi-core CPU's memory hierarchy in the presence of irregular memory access patterns. Further, we have shown that this method extracts the most important contributions to data traffic while only requiring basic hardware characteristics to be specified, such as the size of each cache and its cache lines. Consequently, our method should be applicable to other hardware architectures and cache hierarchies than the multi-core systems we have considered here.

Regarding SpMV, future efforts could use the quantitative performance model presented here to evaluate specific optimisations, such as matrix re-orderings. The data traffic estimates produced by the cache simulation method could potentially be used to tune algorithms and optimisations, though this may place some requirements on how fast such simulations can be carried out. Although we have deliberately focused on a pair of simple and well known SpMV kernels, the presented methods are also applicable to more advanced SpMV algorithms and other irregular computations. For example, one might consider problems related to unstructured meshes, such as the assembly of sparse matrices in finite element methods. These algorithms also suffer from irregular memory accesses, and, moreover, they are often produced through automated code generation and therefore require extensive optimisation. Many relevant optimisations represent trade-offs between different amounts of data traffic and computation, so that more accurate data traffic estimates may help to choose suitable optimisations.

Acknowledgement

This work was supported by the Research Council of Norway under contract 251186. Also, the research presented in this paper has benefited from the Experimental Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053.

References

Agarwal, A., M. Horowitz, and J. Hennessy (May 1989). "An Analytical Cache Model". In: *ACM Transactions on Computer Systems* 7.2, pp. 184–215. ISSN: 0734-2071. DOI: 10.1145/63404.63407.

Aho, A., P. Denning, and J. Ullman (1971). "Principles of Optimal Page Replacement". In: *J. ACM* 18.1, pp. 80–93. ISSN: 1557-735X.

Akbudak, K., E. Kayaaslan, and C. Aykanat (2013). "Hypergraph Partitioning Based Models and Methods for Exploiting Cache Locality in Sparse Matrix-Vector Multiplication". In: *SIAM Journal on Scientific Computing* 35.3, pp. C237–C262. DOI: 10.1137/100813956.

Ballard, G., E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz (2014). "Communication lower bounds and optimal algorithms for numerical linear algebra". In: *Acta Numerica* 23, pp. 1–155. ISSN: 0962-4929.

Bender, M., G. Brodal, R. Fagerberg, R. Jacob, and E. Vicari (2010). "Optimal Sparse Matrix Dense Vector Multiplication in the I/O-Model". In: *Theory of Computing Systems* 47.4, pp. 934–962. ISSN: 1432-4350.

Buluç, A., J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson (2009). "Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks". In: *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*. SPAA '09. Calgary, AB, Canada: ACM, pp. 233–244. ISBN: 978-1-60558-606-9. DOI: 10.1145/1583991.1584053.

Çatalyürek, Ü. V. and C. Aykanat (1999). "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication". eng. In: *Parallel and Distributed Systems, IEEE Transactions on* 10.7, pp. 673–693. ISSN: 1045-9219.

Cruz, R. de la and M. Araya-Polo (2015). "Modeling Stencil Computations on Modern HPC Architectures". In: *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*. Springer International Publishing, pp. 149–171. ISBN: 978-3-319-17248-4.

Davis, T. and Y. Hu (2011). "The University of Florida Sparse Matrix Collection". In: *ACM Transactions on Mathematical Software (TOMS)* 38.1, pp. 1–25. ISSN: 1557-7295.

Eranian, S. and R. Richter (2018). *perfmon2: improving performance monitoring on Linux*. http://perfmon2.sourceforge.net/. Accessed: 2018-11-21.

Filippone, S., V. Cardellini, D. Barbieri, and A. Fanfarillo (Jan. 2017). "Sparse Matrix-Vector Multiplication on GPGPUs". In: *ACM Transactions on Mathematical Software* 43.4, 30:1–30:49. ISSN: 0098-3500. DOI: 10.1145/3017994.

Frigo, M., C. E. Leiserson, H. Prokop, and S. Ramachandran (Jan. 2012). "Cache-Oblivious Algorithms". In: *ACM Transactions on Algorithms* 8.1, 4:1–4:22. ISSN: 1549-6325. DOI: 10.1145/2071379.2071383.

Goumas, G., K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris (2009). "Performance evaluation of the sparse matrix-vector multiplication on modern architectures". In: *The Journal of Supercomputing* 50.1, pp. 36–77. ISSN: 0920-8542.

Haase, G., M. Liebmann, and G. Plank (2007). "A Hilbert-order multiplication scheme for unstructured sparse matrices". In: *International Journal of Parallel, Emergent and Distributed Systems* 22.4, pp. 213–220. DOI: 10.1080/17445760601122084.

Heras, D., V. Blanco, J. Cabaleiro, and F. Rivera (2001). "Modeling and improving locality for the sparse-matrix-vector product on cache memories". In: *Future Generation Computer Systems* 18.1, pp. 55–67. ISSN: 0167-739X.

Intel Corporation (Dec. 2017). *Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*. 325384-065US. Intel Corporation.

— (Apr. 2018). *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 248966-040. Intel Corporation.

Karsavuran, M. O., K. Akbudak, and C. Aykanat (2016). "Locality-Aware Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication on Many-Core Processors". In: *IEEE Transactions on Parallel and Distributed Systems* 27.6, pp. 1713–1726. ISSN: 1045-9219. DOI: 10.1109/TPDS.2015.2453970.

Kreutzer, M., G. Hager, G. Wellein, H. Fehske, and A. R. Bishop (2014). "A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units". In: *SIAM Journal on Scientific Computing* 36.5, pp. 401–423. ISSN: 1064-8275.

Langguth, J., N. Wu, J. Chai, and X. Cai (2015a). "Parallel performance modeling of irregular applications in cell-centered finite volume methods over unstructured tetrahedral meshes". In: *Journal of Parallel and Distributed Computing* 76, pp. 120–131. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.10.005.

Langguth, J., M. Sourouri, G. T. Lines, S. B. Baden, and X. Cai (July 2015b). "Scalable Heterogeneous CPU-GPU Computations for Unstructured Tetrahedral Meshes". In: *IEEE Micro* 35.4, pp. 6–15. ISSN: 0272-1732. DOI: 10.1109/MM.2015.70.

Liu, W. and B. Vinter (2015). "CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication". In: *Proceedings of the 29th ACM on international conference on supercomputing*. ICS '15. ACM, pp. 339–350. ISBN: 978-1-4503-3559-1.

Liu, X., M. Smelyanskiy, E. Chow, and P. Dubey (2013). "Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors". In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ICS '13. ACM, pp. 273–282. ISBN: 978-1-4503-2130-3. DOI: 10.1145/2464996.2465013.

Malossi, A. C. I., Y. Ineichen, C. Bekas, A. Curioni, and E. S. Quintana-Orti (2014). "Performance and Energy-Aware Characterization of the Sparse Matrix-Vector Multiplication on Multithreaded Architectures". In: *2014 43rd International Conference on Parallel Processing Workshops*. IEEE, pp. 139–148. ISBN: 9781479956159.

McCalpin, J. D. (Dec. 1995). "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25.

Merrill, D. and M. Garland (2016). "Merge-based parallel sparse matrix-vector multiplication". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '16. IEEE Press, pp. 1–12. ISBN: 9781467388153.

Molka, D., R. Schöne, D. Hackenberg, and W. Nagel (2017). "Detecting Memory-Boundedness with Hardware Performance Counters". In: *Proceedings of the 8th ACM/SPEC on international conference on performance engineering*. ICPE '17. ACM, pp. 27–38. ISBN: 9781450344043.

Nishtala, R., R. W. Vuduc, J. W. Demmel, and K. A. Yelick (May 2007). "When cache blocking of sparse matrix vector multiply works and why". In: *Applicable Algebra in Engineering, Communication and Computing* 18.3, pp. 297–311. ISSN: 1432-0622. DOI: 10.1007/s00200-007-0038-9.

Pinar, A. and M. T. Heath (1999). "Improving Performance of Sparse Matrix-vector Multiplication". In: *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*. SC '99. Portland, Oregon, USA: ACM. ISBN: 1-58113-091-0. DOI: 10.1145/331532.331562.

Saad, Y. (2003). *Iterative methods for sparse linear systems*. 2nd ed. SIAM. ISBN: 978-0-898715-34-7.

Sandberg, A., D. Black-Schaffer, and H. Erik (2011). "A simple statistical cache sharing model for multicores". eng. In: *Proc. 4th Swedish Workshop On Multi-Core Computing*, pp. 31–36.

Stengel, H., J. Treibig, G. Hager, and G. Wellein (2015). "Quantifying Performance Bottlenecks of Stencil Computations Using the Execution-Cache-Memory Model". In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. ICS '15. Newport Beach, California, USA: ACM, pp. 207–216. ISBN: 978-1-4503-3559-1. DOI: 10.1145/2751205.2751240.

Temam, O. and W. Jalby (1992). "Characterizing the Behavior of Sparse Algorithms on Caches". In: *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*. Supercomputing '92. Minneapolis, Minnesota, USA: IEEE Computer Society Press, pp. 578–587. ISBN: 0-8186-2630-5.

Toledo, S. (1997). "Improving the memory-system performance of sparse-matrix vector multiplication". In: *IBM Journal of Research and Development* 41.6, pp. 711–725.

Uhlig, R. A. and T. N. Mudge (June 1997). "Trace-driven Memory Simulation: A Survey". In: *ACM Computing Surveys* 29.2, pp. 128–170. ISSN: 0360-0300. DOI: 10.1145/254180.254184.

Vastenhouw, B. and R. H. Bisseling (2005). "A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication". In: *SIAM Review* 47.1, pp. 67–95. DOI: 10.1137/S0036144502409019.

Vuduc, R. W. (Jan. 2004). "Automatic performance tuning of sparse matrix kernels". PhD thesis. Berkeley, CA, USA: University of California. URL: http://bebop.cs.berkeley.edu/pubs/vuduc2003-dissertation.pdf.

Vuduc, R. W., J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee (2002). "Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply". In:

Proceedings of the 2002 ACM/IEEE Conference on Supercomputing. SC '02. Baltimore, Maryland: IEEE Computer Society Press, pp. 1–35. ISBN: 0-7695-1524-X.

Willcock, J. and A. Lumsdaine (2006). "Accelerating Sparse Matrix Computations via Data Compression". In: *Proceedings of the 20th Annual International Conference on Supercomputing*. ICS '06. Cairns, Queensland, Australia: ACM, pp. 307–316. ISBN: 1-59593-282-8. DOI: 10.1145/1183401.1183444.

Williams, S., L. Oliker, R. W. Vuduc, J. Shalf, K. Yelick, and J. Demmel (2009a). "Optimization of sparse matrix-vector multiplication on emerging multicore platforms". In: *Parallel Computing* 35.3, pp. 178–194. ISSN: 0167-8191. DOI: 10.1016/j.parco.2008.12.006.

Williams, S., A. Waterman, and D. Patterson (Apr. 2009b). "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Communications of the ACM* 52.4, pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785.

Yzelman, A. N. and R. H. Bisseling (2009). "Cache-Oblivious Sparse Matrix-Vector Multiplication by Using Sparse Matrix Partitioning Methods". In: *SIAM Journal on Scientific Computing* 31.4, pp. 3128–3154. DOI: 10.1137/080733243.

— (2011). "Two-dimensional cache-oblivious sparse matrix-vector multiplication". In: *Parallel Computing* 37.12, pp. 806–819. ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.08.004.

— (2012). "A Cache-Oblivious Sparse Matrix–Vector Multiplication Scheme Based on the Hilbert Curve". In: *Progress in Industrial Mathematics at ECMI 2010*. Ed. by M. Günther, A. Bartel, M. Brunk, S. Schöps, and M. Striebel. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 627–633. ISBN: 978-3-642-25100-9.

Yzelman, A. N. and D. Roose (2014). "High-Level Strategies for Parallel Shared-Memory Sparse Matrix-Vector Multiplication". In: *IEEE Transactions on Parallel and Distributed Systems* 25.1, pp. 116–125. ISSN: 1045-9219. DOI: 10.1109/TPDS.2013.31.

Zhang, W. and X. Cai (2016). "Solving 3D Time-Fractional Diffusion Equations by High-Performance Parallel Computing". In: *Fractional Calculus and Applied Analysis* 19.1, pp. 140–160. ISSN: 1311-0454. DOI: 10.1515/fca-2016-0008.

Paper II

On memory traffic and optimisations for low-order finite element assembly algorithms on multi-core CPUs

James D. Trotter, Simula Research Laboratory and University of Oslo, Norway
Xing Cai, Simula Research Laboratory and University of Oslo, Norway
Simon W. Funke, Simula Research Laboratory

Submitted for publication.

Abstract

Motivated by the wish to understand the achievable performance of finite element assembly on unstructured computational meshes, we dissect the standard cellwise assembly algorithm into four kernels, two of which are dominated by irregular memory traffic. Several optimisation schemes are studied together with associated lower and upper bounds on the estimated memory traffic volume. Apart from properly reordering the mesh entities, the two most significant optimisations include adopting a lookup table in adding element matrices or vectors to their global counterparts, and using a rowwise assembly algorithm for multi-threaded parallelisation. Rigorous benchmarking shows that, due to the various optimisations, the actual volumes of memory traffic are in many cases very close to the estimated lower bounds. These results confirm the effectiveness of the optimisations, while also providing a recipe for developing efficient software for finite element assembly.

1 Introduction

Finite element methods are among the most important and widely used techniques for numerically solving partial differential equations (PDEs), especially when the geometry of the solution domain is best described by an *unstructured* computational mesh. While offering flexibility to handle complicated geometries, unstructured meshes bring challenges with respect to achieving performance on modern computing hardware. In particular, unstructured meshes lead to irregular memory access patterns that are difficult for hierarchical, cache-based memories. This is due to poor data prefetching and limited data reuse in the caches. Another challenge arises with multi-threaded parallel computing because irregular memory accesses can lead to race conditions (i.e., several threads simultaneously update the same variables) that are not easy to determine beforehand.

From a computational point of view, the application of finite element methods to PDEs ultimately translates into assembling systems of linear algebraic equations and then solving them. In many cases, a significant portion of the overall computation

occurs during the assembly stage, which is the topic of the current paper. Our aim is to understand how *memory traffic*—the movement of data to and from memory—impacts the performance of such calculations in connection with unstructured meshes. Generally speaking, it is important to execute the assembly with high parallel performance to prevent it from becoming a bottleneck. For non-linear problems, assembly is also critical to achieving good performance, because it must be carried out during each iteration of a non-linear solver.

Considerable efforts have gone into optimising finite element assembly algorithms [Cantwell et al. 2011; Kronbichler and Kormann 2012; Markall et al. 2013; Vos et al. 2010], particularly in connection with automatically generating high-performance assembly code [Kirby et al. 2005; Kirby and Logg 2006; Luporini et al. 2017; Luporini et al. 2015; Ølgaard and Wells 2010; Russell and Kelly 2013; Sun et al. 2020]. However, the emphasis of these efforts is mainly placed on reducing the number of floating-point operations performed, without fully accounting for the impact of memory traffic on the performance. For discretisations that are based on low-degree polynomials, such as linear and quadratic finite elements, counting floating-point operations alone is not sufficient to understand the achievable performance. Instead, it is important in these cases to have a quantified understanding of the memory traffic and devise optimisation strategies accordingly.

In this paper, our initial focus is on the standard cellwise assembly algorithm, which is commonly used for building vectors and sparse matrices from finite element variational forms. We dissect this algorithm into four different kernels and investigate them in detail with respect to the floating-point computations and memory traffic involved. The novelty of our work is the derivation of precise lower and upper bounds on memory traffic for the memory-traffic-heavy kernels and several optimisation strategies. These optimisations even include two seemingly memorywise wasteful schemes, i.e., use of a lookup table in adding element matrices and vectors to their global counterparts, and a rowwise assembly algorithm. The latter is for avoiding race conditions in the context of multi-threaded parallel assembly. Our contribution includes a quantification of the effectiveness of the optimisations, by rigorously benchmarking the actual memory traffic volumes (measured by hardware performance counters) against the lower and upper bounds. Our work thus provides a recipe for developing serial and parallel software for low-order finite element asembly algorithms.

The remainder of the paper is organised as follows. We begin with some background information on finite element assembly in Section 2. The cellwise assembly algorithm is detailedly dissected in Section 3, whereas a few memory-oriented optimisations, including the rowwise assembly algorithm, are explored in Section 4. In Section 5, we derive precise lower and upper bounds on the memory traffic involved in these calculations. Next, Section 6 presents benchmarking results from various finite element assembly computations. In Section 7, connections are made to the relevant work on finite element methods, before we draw our conclusions in Section 8.

2 Background

In this section, we briefly present some notation and familiar concepts from the finite element literature that are needed to describe the usual finite element assembly procedure. Further details can be found, for instance, in [Ciarlet 2002; Ern and Guermond 2004].

Computations based on finite elements usually begin with a variational formulation, consisting of a bilinear form $a\colon V \times U \to \mathbf{R}$ and a linear form $\ell\colon V \to \mathbf{R}$, where V and U are called, respectively, the test and trial function spaces. The test and trial space are often one and the same, though, in general, they do not need to be. The objective is to find a solution $u \in U$ such that $a(v,u) = \ell(v)$ for every $v \in V$. If we choose a basis $\{\psi_i\}_{i=0}^{M-1}$ for the test space and another basis $\{\phi_j\}_{j=0}^{N-1}$ for the trial space, then we can compute a linear system of equations, $Ax = b$, with the coefficient matrix $A_{i,j} = a(\psi_i, \phi_j)$ and right-hand side vector $b_i = \ell(\psi_i)$. When the linear system is solved, it yields a solution $u = \sum_{j=0}^{N-1} x_j \phi_j$ to the variational formulation of the PDE.

The bilinear form a and linear form ℓ are primarily defined in terms of integrals over a domain $\Omega \subset \mathbf{R}^d$, $d > 0$, though some additional constraints may arise due to handling prescribed boundary conditions. To illustrate, we use Poisson's equation as the classical example of a linear elliptic PDE,

$$-\triangle u = f \quad \text{in } \Omega$$

for some $f \in L^2(\Omega)$ and the homogeneous Neumann boundary condition. In this case, the test and trial spaces are finite-dimensional subspaces of the Sobolev space $H^1(\Omega)$, and the standard variational form consists of the weak formulation of the Laplacian

$$a(v,u) = \int_\Omega \nabla v \cdot \nabla u \,\mathrm{d}x,$$

and

$$\ell(v) = \int_\Omega f v \,\mathrm{d}x.$$

The basic approach, as outlined above, also applies to systems of PDEs, where the trial and test spaces consist of vector-valued functions, and to time-dependent and non-linear problems, both of which are reduced to solving sequences of linear problems.

In this paper, we are interested in the assembly of coefficient matrices that are derived from bilinear variational forms whose test and trial spaces are defined in terms of finite elements. First, a computational mesh, or triangulation, denoted by $\mathcal{T}$, is introduced by partitioning a domain Ω into cells, usually convex polygons or polyhedra. Next, local function spaces $X(T)$ on each mesh cell $T \in \mathcal{T}$ are combined to form a global finite element space $X(\mathcal{T})$, consisting of functions defined piecewise with respect to the mesh $\mathcal{T}$. Here, we use $\theta_0, \theta_1, \ldots, \theta_{M-1}$ to denote the basis functions of the global finite element space $X(\mathcal{T})$. For each mesh cell T, there is an injective map $\mu\colon [0,m) \to [0,M)$, called a local-to-global mapping, that identifies each basis function $\theta_0^T, \theta_1^T, \ldots, \theta_{m-1}^T$ of the local space $X(T)$ with a corresponding global basis function, such that $\theta_{\mu(j)}(x) = \theta_j^T(x)$ whenever $x \in T$, for $0 \le j < m$. For simplicity, we restrict

for all cells $T \in \mathcal{T}$ **do**
1. Gather vertex coordinates of T
2. Transform to a reference cell $\hat{T}$
3. Compute element matrix A_T
4. Scatter results to the global matrix A

end for

Algorithm 1. Cellwise finite element assembly.

our attention to polynomial spaces and the commonly used Lagrange elements, whose basis consists of Lagrange interpolating polynomials on each mesh cell.

The most common approach is to use a cellwise assembly algorithm that computes the matrix A as a sum over the mesh cells,

$$A = \sum_{T \in \mathcal{T}} P_T A_T Q_T^{\mathsf{T}}. \tag{1}$$

Denoting $m = \dim V(T)$ and $n = \dim U(T)$, then $A_T \in \mathbf{R}^{m,n}$ is an element matrix for a mesh cell T,

$$(A_T)_{i,j} = a(\tilde{\psi}_i, \tilde{\phi}_j),$$

where $\tilde{\psi}_i \in V(T)$ and $\tilde{\phi}_j \in U(T)$ are local basis functions. The purpose of the matrices $P_T \in \mathbf{R}^{M,m}$ and $Q_T \in \mathbf{R}^{N,n}$ is to "scatter" contributions from the element matrix A_T to the correct locations of the global matrix A. More specifically, if μ and ν denote local-to-global mappings of the test and trial space, respectively, then $(P_T)_{i,j} = \delta_{i,\mu(j)}$ and $(Q_T)_{i,j} = \delta_{i,\nu(j)}$, where $\delta_{i,j}$ denotes the Kronecker delta function.

3 Cellwise finite element assembly

For a global matrix A corresponding to some bilinear variational form and a mesh $\mathcal{T}$, the high-level pseudo-code in Algorithm 1 describes a typical cellwise assembly algorithm. The calculations pertaining to each mesh cell can thus be dissected into four separate parts, or kernels, each of which is described in detail in the following subsections.

3.1 Gathering vertex coordinates

The first step is to gather the coordinates of a cell's vertices, from a data structure storing the entire mesh $\mathcal{T}$, before they can be used to compute the transformation to a reference cell and the element matrix. In general, an unstructured mesh is used, thus an explicit representation of the connectivity between mesh cells is needed. In this paper, we primarily consider three-dimensional meshes whose cells are tetrahedra, though the following text extends easily to other commonly used mesh cells, such as quadrilaterals or hexahedra.

Let $\Omega \subset \mathbf{R}^d$ ($d = 3$) denote a bounded, polyhedral domain, and let $\mathcal{T}$ be a mesh with vertices $p_0, p_1, \ldots, p_{M-1} \in \mathbf{R}^d$ that partitions Ω into the d-dimensional mesh cells,

```
void simplex3_gather_vertex_coordinates(
  const double * restrict vertex_coords,
  const int * restrict vertex_indices,
  int cell_index,
  double cell_vertex_coords[12])
{
  for (int i = 0; i < 4; i++) {
    int v_id = vertex_indices[cell_index*4+i];
    for (int j = 0; j < 3; j++) {
      cell_vertex_coords[i*3+j] =
        vertex_coords[v_id*3+j];
    }
  }
}
```

Algorithm 2. Gathering the vertex coordinates of a tetrahedron.

$T_0, T_1, \ldots, T_{N-1}$. Each cell T is defined by m distinct vertices, $p_{\sigma(0)}, p_{\sigma(1)}, \ldots, p_{\sigma(m-1)}$, where $\sigma\colon [0, m) \to [0, M)$ is an injective map.

If we consider all the vertex coordinates of the mesh $\mathcal{T}$ as an $M \times d$ matrix,

$$\begin{bmatrix} p_{0,0} & p_{0,1} & \cdots & p_{0,d-1} \\ p_{1,0} & p_{1,1} & \cdots & p_{1,d-1} \\ \vdots & \vdots & \ddots & \vdots \\ p_{M-1,0} & p_{M-1,1} & \cdots & p_{M-1,d-1} \end{bmatrix},$$

then these values are typically stored in memory in a row-major order. The vertex coordinates are usually not accessed in sequence. Instead, gathering the coordinates of a cell $T = \begin{bmatrix} p_{\sigma(0)} & \ldots & p_{\sigma(m-1)} \end{bmatrix}$ requires loading m different rows, $\sigma(0), \ldots, \sigma(m-1)$, each with d consecutive values.

For the typical case of a tetrahedral mesh, twelve values are loaded per T, because each cell has four vertices, and each vertex has three coordinates. The code in Algorithm 2 shows how the vertex coordinates of a cell may be gathered in this case. Note that the `restrict` keyword is used as a hint to the compiler that accesses through the different pointers will not refer to the same memory locations, thereby allowing the compiler to apply certain optimisations that it otherwise might not.

3.2 Transforming to a reference cell

Since the element matrices are computed by integration, it is customary to first perform a change of variables that shifts the domain of integration from a given cell $T \in \mathcal{T}$ to a reference cell $\hat{T} \subset \mathbf{R}^d$. The change of variables is needed to apply standard numerical integration schemes, which are defined for integrals over certain reference domains, such as the standard simplex or unit cube. In addition, some parts of the integrals can often be simplified or precomputed, because they are independent of the mesh geometry.

Given a mesh cell $T \in \mathcal{T}$ and a reference cell $\hat{T} \subset \mathbf{R}^d$, let $F\colon \hat{T} \to T$ be a one-to-one C^1-mapping, such that $F'(\hat{x}) \neq 0$ for all $\hat{x} \in \hat{T}$, where $F'(\hat{x})$ is the Jacobian of F, i.e., $\big(F'(\hat{x})\big)_{i,j} = \frac{\partial F_i}{\partial \hat{x}_j}(\hat{x})$. In general, for a continuous function $f\colon \mathbf{R}^d \to \mathbf{R}$, whose support is compact and lies in T, a change of variables from T to a reference cell $\hat{T}$ gives

$$\int_T f(x)\,\mathrm{d}x = \int_{\hat{T}} f\big(F(\hat{x})\big)\big|\det F'(\hat{x})\big|\,\mathrm{d}\hat{x}.$$

In the case of a d-dimensional simplicial mesh, the reference cell is usually the standard simplex, $\hat{T} = \begin{bmatrix} e_0 & e_1 & \cdots & e_{d-1} & 0 \end{bmatrix}$, where $e_i \in \mathbf{R}^d$ is the i-th unit vector, $0 \leq i < d$, and $0 \in \mathbf{R}^d$ is the zero vector. If T is a d-dimensional simplex with vertices $p_0, p_1, \ldots, p_d \in \mathbf{R}^d$, then an affine mapping from the standard simplex to T is defined by

$$F(\hat{x}) = p_d + \sum_{k=0}^{d-1} \hat{x}_k (p_k - p_d). \tag{2}$$

The Jacobian of this mapping is

$$\big(F'(\hat{x})\big)_{i,j} = (p_j - p_d) \cdot e_i = p_{j,i} - p_{d,i}, \tag{3}$$

which, it may be noted, is constant over $\hat{T}$ and does not depend on $\hat{x}$. Furthermore, the inverse of the Jacobian $\big(F'(\hat{x})\big)^{-1}$ is often required and can be computed from Cramer's rule,

$$\big(F'(\hat{x})\big)^{-1} = \frac{1}{\det F'(\hat{x})} C_{F'}^{\mathsf{T}},$$

where $C_{F'}$ is the matrix of cofactors of $F'(\hat{x})$. Recall that the (i, j)-th cofactor of a matrix is computed from the determinant of the sub-matrix obtained by omitting the i-th row and j-th column and multiplying by a factor $(-1)^{i+j}$.

For a tetrahedral mesh, the code in Algorithm 3 shows how the transformation to the reference tetrahedron is computed, requiring a total of 41 floating-point operations.

3.3 Computing the element matrix

Once a cell's vertex coordinates and the mapping to a reference cell have been obtained, the element matrix can be computed. This kernel must be tailored to an individual variational form. Alternatively, projects such as FEniCS [Logg et al. 2012] and Firedrake [Rathgeber et al. 2016] can ease the burden of hand-crafting these kernels by automatically generating code that will compute element matrices based on high-level descriptions of variational forms.

To illustrate, let us consider the variational form for the Laplacian with respect to a test space $V(\mathcal{T})$ and trial space $U(\mathcal{T})$,

$$a(v, u) = \int_\Omega \nabla v \cdot \nabla u\,\mathrm{d}x.$$

In this example, the element matrix A_T is

$$\big(A_T\big)_{i,j} = \int_T \nabla \psi_i \cdot \nabla \phi_j\,\mathrm{d}x,$$

```
void simplex3_transform_to_reference_cell(
  const double p[12],      // Vertex coords
  double J[9],             // Jacobian matrix
  double C[9],             // Cofactors of J
  double * restrict J_det) // Determinant of J
{
  // Compute Jacobian matrix
  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
      J[i*3+j] = p[j*3+i] - p[3*3+i];
    }
  }

  // Compute cofactors of the Jacobian
  C[0] = J[4]*J[8]-J[5]*J[7];
  C[1] = J[5]*J[6]-J[3]*J[8];
  C[2] = J[3]*J[7]-J[4]*J[6];
  C[3] = J[2]*J[7]-J[1]*J[8];
  C[4] = J[0]*J[8]-J[2]*J[6];
  C[5] = J[1]*J[6]-J[0]*J[7];
  C[6] = J[1]*J[5]-J[2]*J[4];
  C[7] = J[2]*J[3]-J[0]*J[5];
  C[8] = J[0]*J[4]-J[1]*J[3];

  // Compute determinant of the Jacobian
  *J_det = J[0]*C[0]+J[1]*C[1]+J[2]*C[2];
}
```

Algorithm 3. Transforming an arbitrary tetrahedron to the reference tetrahedron. The 3×3 Jacobian matrix J is computed using Eq. (3). The cofactors of the Jacobian matrix are stored in C, where J_det stores the determinant of the Jacobian.

for local basis functions $\psi_i \in V(T)$ and $\phi_j \in U(T)$. Applying a change of variables, based on a mapping $F\colon \hat{T} \to T$ for a reference cell $\hat{T}$, leads to

$$\left(A_T\right)_{i,j} = \int_{\hat{T}} \left(F'(\hat{x})\right)^{-\mathsf{T}} \nabla\hat{\psi}_i \cdot \left(F'(\hat{x})\right)^{-\mathsf{T}} \nabla\hat{\phi}_j \left|\det F'(\hat{x})\right| \mathrm{d}\hat{x},$$

where $\hat{\psi}_i = \psi_i \circ F$ and $\hat{\phi}_j = \phi_j \circ F$. Note that $\hat{\psi}_i$ and $\hat{\phi}_j$ are local basis functions for $V(\hat{T})$ and $U(\hat{T})$, respectively.

In some cases, integrals are computed by using a numerical integration scheme, whereby the integrand is evaluated at a number of quadrature points, multiplied by suitable weights, and then the results are added together. However, in many cases, element matrices can be rewritten into a tensor representation [Kirby and Logg 2006] by factoring out terms that are independent of the cell geometry. More specifically, the element matrix is written as a tensor contraction $A_T = \sum_\alpha \hat{A}_\alpha G_T^\alpha$, where $\hat{A}$ is a reference tensor and G_T is a geometry tensor that depends on the mesh cell T. The

advantage is that integrals appearing in the reference tensor can be precomputed and do not need to be evaluated at runtime. This strategy is employed by FEniCS in its automated generation of kernels for computing element matrices.

For example, the element matrix for the Laplacian can be written as the contraction of a reference tensor $\hat{A}$ of rank four and a geometry tensor G_T of rank two. More specifically,

$$\hat{A}_{i,j,k_0,k_1} = \int_{\hat{T}} \frac{\partial \hat{\psi}_i}{\partial x_{k_0}} \frac{\partial \hat{\phi}_j}{\partial x_{k_1}} \,\mathrm{d}\hat{x}, \tag{4}$$

for $0 \leq i < \dim V(\hat{T})$, $0 \leq j < \dim U(\hat{T})$, and

$$G_T^{k_0,k_1} = \frac{1}{\left|\det F'\right|} \sum_{l=0}^{d-1} \left(C_{F'}\right)_{l,k_0} \left(C_{F'}\right)_{l,k_1}, \tag{5}$$

for $0 \leq k_0, k_1 < d$. Because the basis functions with respect to the reference element are known beforehand, the reference tensor can be precomputed. Algorithm 4 shows a kernel that uses this approach to compute element matrices for the Laplacian with first-order Lagrange elements on tetrahedra. The kernel for second-order elements is similar, though additional basis functions result in a 10×10 element matrix.

3.4 Scattering results to a global matrix

In the final stage of the cellwise assembly, the contributions from an element matrix A_T are added to the global matrix A. Let $V(T)$ and $U(T)$ denote local finite element spaces with local-to-global mappings $\mu\colon [0, m) \to [0, M)$ and $\nu\colon [0, n) \to [0, N)$, respectively. Then, for each pair of local basis functions, (ψ_i, ϕ_j), for $0 \leq i < m$ and $0 \leq j < n$, the corresponding value of the global matrix is updated,

$$A_{\mu(i),\nu(j)} \leftarrow A_{\mu(i),\nu(j)} + \left(A_T\right)_{i,j}.$$

However, because the global matrix A is sparse with only its non-zeros stored, it is necessary to locate the correct position in the array that stores the non-zero values of A before each update.

We assume that the global matrix A is stored in the commonly used format of *compressed sparse row* (CSR). Suppose there are in total K non-zero entries $a_0, a_1, \ldots, a_{K-1}$, arranged in an ascending order according to their row and column indices (i_k, j_k), $0 \leq k < K$. Then, in addition to the K non-zeros, we store the K column indices $j_0, j_1, \ldots, j_{K-1}$ and $(M+1)$ row pointers $r_0, r_1, \ldots, r_M$. The row pointers r_i and $r_{i+1} - 1$ are the indices of the first and last non-zeros of the i-th row, respectively. In other words, a non-zero a_k belongs to the i-th row if $r_i \leq k < r_{i+1}$.

Now, it remains to find the non-zero matrix entry a_k that corresponds to the global matrix value $A_{\mu(i),\nu(j)}$. For a CSR matrix, the row pointers are used to find matrix entries that belong to the given row, i.e., $r_{\mu(i)} \leq k < r_{\mu(i)+1}$. Because the non-zeros within each row are sorted by their column indices, a common strategy is to perform a binary search to find the non-zero matrix entry a_k whose column index equals $\nu(j)$. Algorithm 5 shows how this is done when the test and trial spaces are first-order Lagrange elements on tetrahedra.

```
void laplacian_simplex3_p1_element_matrix(
  const double J[9],   // Jacobian matrix
  const double C[9],   // Cofactors of J
  double J_det,        // Determinant of J
  double * restrict A) // Element matrix
{
  // Compute 3-by-3 symmetric geometry tensor
  double G[6];
  double d = (1./6.) / fabs(J_det);
  G[0] = d*(C[0]*C[0]+C[3]*C[3]+C[6]*C[6]);
  G[1] = d*(C[0]*C[1]+C[3]*C[4]+C[6]*C[7]);
  G[2] = d*(C[0]*C[2]+C[3]*C[5]+C[6]*C[8]);
  G[3] = d*(C[1]*C[1]+C[4]*C[4]+C[7]*C[7]);
  G[4] = d*(C[1]*C[2]+C[4]*C[5]+C[7]*C[8]);
  G[5] = d*(C[2]*C[2]+C[5]*C[5]+C[8]*C[8]);

  // Compute 4-by-4 element matrix
  A[0]=G[0]; A[1]=G[1]; A[ 2]=G[2];
  A[4]=G[1]; A[5]=G[3]; A[ 6]=G[4];
  A[8]=G[2]; A[9]=G[4]; A[10]=G[5];
  A[ 3]=A[12]=-G[0]-G[1]-G[2];
  A[ 7]=A[13]=-G[1]-G[3]-G[4];
  A[11]=A[14]=-G[2]-G[4]-G[5];
  A[15]=G[0]+2*G[1]+2*G[2]+G[3]+2*G[4]+G[5];
}
```

Algorithm 4. Computing an element matrix for the Laplacian with test and trial spaces of first-order Lagrange elements on tetrahedra. The element matrix is computed as a tensor contraction of a reference tensor and a geometry tensor given by Eq. (4) and Eq. (5), respectively. The geometry tensor is a symmetric 3×3 matrix, computed from the Jacobian determinant `J_det` and the cofactors `C`, and stored in the array `G`. The reference tensor is independent of the mesh geometry, so its values are hard-coded into the computation.

4 Optimisations

In this section, we consider some optimisations that are relevant for the finite element assembly algorithm described in Section 3.

4.1 Reordering an unstructured mesh

The arithmetic intensity of finite element assembly algorithms is low when low-order Lagrange elements are used. Therefore, it is prudent to consider optimisations that improve data locality and use caches more effectively.

The memory access patterns and data locality associated with gathering the vertex coordinates of each mesh cell (see Algorithm 2) depends on the order in which mesh cells are visited. Therefore, reordering the vertices and cells of an unstructured mesh, prior to the finite element assembly, can yield improved performance. For similar pur-

```
void scatter_to_global_matrix_bsearch(
  const int test_space_dofs[4],
  const int trial_space_dofs[4],
  const double element_matrix[16],
  const int * row_ptr,
  const int * col_idx,
  double * values)
{
  for (int i = 0; i < 4; i++) {
    int row = test_space_dofs[i];
    for (int j = 0; j < 4; j++) {
      int col = trial_space_dofs[j];
      int p = row_ptr[row];
      int q = row_ptr[row+1]-1;
      int r = p;
      while (p <= q) {
        r = (p+q) / 2;
        if (col_idx[r] == col) break;
        else if (col_idx[r] < col) p = r+1;
        else q = r-1;
      }
      values[r] += element_matrix[i*4+j];
    }
  }
}
```

Algorithm 5. Scattering a 4×4 element matrix to a global matrix stored in CSR format. The location of each corresponding non-zero in the global matrix is found by a binary search among the non-zeros of the respective row.

poses, researchers have previously studied cache-efficient mesh layouts, for instance, to improve the performance of visualising unstructured meshes [Tchiboukdjian et al. 2010; Tchiboukdjian et al. 2008; Yoon and Lindstrom 2006].

In this paper, we opt for a simple approach that is commonly used in connection with finite element methods. First, we reorder the vertices of a mesh based on the Reverse Cuthill-McKee (RCM) algorithm [Cuthill and McKee 1969]. Specifically, we apply the RCM algorithm to an adjacency matrix whose rows and columns correspond to the mesh vertices and whose entries indicate whether or not a pair of vertices have a mesh cell in common. Roughly speaking, this method leads to vertices being closer to each other whenever they belong to the same mesh cells.

It is important to also reorder the cells of a mesh, because a good vertex ordering alone does not imply good data locality when gathering the vertex coordinates of each mesh cell. We thus rearrange the mesh cells in a lexicographic order according to their vertex indices. This simple technique appears to be quite effective in practice.

4.2 Reordering global degrees of freedom

When scattering element matrices to a global matrix in the CSR format (see Algorithm 5), the numbering of the global degrees of freedom greatly impacts the memory access pattern and data locality. Generally speaking, reordering the global degrees of freedom of the test and trial spaces is equivalent to reordering the equations and unknowns of the assembled linear system, and this is a well-known method for improving performance of sparse linear solvers, particularly direct solvers. In fact, there exists a variety of sparse matrix reorderings that are commonly used in connection with finite element methods [George 1973; George and Mcintyre 1978], including the RCM algorithm [Cuthill and McKee 1969] that we use to reorder the vertices of a mesh.

In fact, for first-order Lagrange elements, the global degrees of freedom are usually numbered in the same way as the mesh vertices. Thus, reordering the mesh vertices is equivalent to reordering the global degrees of freedom. For second- and higher-order elements, it is also possible to apply the RCM algorithm to reorder the global degrees of freedom. However, in this paper, we instead order the global degrees of freedom according to the mesh entities that they belong to. In this way, it is sufficient to choose an ordering for the faces of the mesh, and the global degrees of freedom will be numbered accordingly. Like the mesh cells, we order the faces of a mesh lexicographically according to their vertex indices.

4.3 Using a lookup table in the scattering kernel

Another concern that affects the performance of scattering element matrices to a global matrix is the binary search (Algorithm 5) that is performed for each entry of an element matrix. This is needed to find the location within the CSR structure of the corresponding non-zero entry in the global matrix. As an alternative, it is possible to precompute and store the locations of the global matrix non-zeros for each entry of every element matrix. The non-zero locations can usually be obtained for free or at little extra cost when the sparsity pattern of the global matrix is computed prior to the assembly. The lookup table can then be reused every time the matrix is assembled.

Although this approach requires a slight increase in storage requirements and memory traffic to accommodate the lookup table, the benefit is that the binary searches, which are potentially quite costly, can now be omitted. An example of this approach is shown in Algorithm 6, which should be contrasted with Algorithm 5.

4.4 Cross-element vectorisation

For kernels that require some amount of floating-point calculations, it is often necessary to employ vectorisation to make the most effective use of a multi-core CPU. For finite element assembly, proper use of vectorisation can be used to speed up the calculation of element matrices. Although the compiler is sometimes able to automatically vectorise, there are cases that require such optimisations to be applied manually, for instance, using a technique known as cross-element vectorisation [Sun et al. 2020]. However, we note that the kernels we consider here are both simple and lightweight in terms of floating-point calculations, so we do not expect vectorisation to have much of an impact

```
void scatter_to_global_matrix_lookup(
  const int nonzero_locations[16],
  const double element_matrix[16],
  double * values)
{
  for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
      int k = nonzero_locations[i*4+j];
      values[k] += element_matrix[i*4+j];
    }
  }
}
```

Algorithm 6. Scattering a 4×4 local matrix to a global matrix in CSR format. A lookup table `nonzero_locations` is used to find the correct locations of the non-zero values.

on the overall assembly time. We include it nonetheless for the sake of completeness and because it may play a more important role for more advanced kernels.

The idea behind cross-element vectorisation is to compute element matrices of several cells simultaneously by letting each vector lane correspond to a different mesh cell. This requires rearranging the input data, i.e., the vertex coordinates of the cells, after being brought into the first-level cache. Because vector operations are primarily designed to act on a sequence of contiguous memory locations, the vertex coordinates of each cell are interleaved. Once the data has been rearranged, the element matrix computation is almost identical to the scalar kernel. The code in Algorithm 7 shows how element matrices for the Laplacian with first-order Lagrange elements may be computed using cross-element vectorisation and AVX512 intrinsics.

4.5 Rowwise assembly

The most common strategy for parallelising a cellwise finite element assembly in shared memory, using multiple threads, is to assign a roughly equal number of cells to each thread. There is an inherent difficulty because the mesh cells that share global basis functions may be assigned to different threads. To avoid incorrect results of the scattering kernel, one must take care to prevent race conditions, e.g., by introducing atomic operations or another form of synchronisation.

An alternative to the usual cellwise assembly algorithm is to compute an entire row of the global matrix at a time. From Eq. (1), we find that the i-th row and j-th column of the global matrix A can be expressed

$$A_{i,j} = \sum_{T \in \mathcal{T}} \left(P_T A_T Q_T^\mathsf{T}\right)_{i,j} = \sum_{T \in \mathcal{T}} \sum_{k=0}^{m-1} \delta_{i,\mu(k)} \left(A_T Q_T^\mathsf{T}\right)_{k,j},$$

where μ is a local-to-global mapping for the test space $V(\mathcal{T})$. Recall that the i-th row of the matrix corresponds to a global basis function $\theta_i \in V(\mathcal{T})$, and this global basis function is associated with a face f_i of the mesh $\mathcal{T}$. Moreover, for a mesh cell T, we

```
void laplacian_simplex3x8_p1_element_matrix(
  const __m512d J[9],   // Jacobian matrices
  const __m512d C[9],   // Cofactors of J
  __m512d J_det,        // Determinants of J
  __m512d * restrict A) // Element matrices
{
  // Compute 3-by-3 symmetric geometry tensors
  __m512d G[6];
  __m512d d = (1./6.) / _mm512_abs_pd(J_det);
  G[0] = d*(C[0]*C[0]+C[3]*C[3]+C[6]*C[6]);
  G[1] = d*(C[0]*C[1]+C[3]*C[4]+C[6]*C[7]);
  G[2] = d*(C[0]*C[2]+C[3]*C[5]+C[6]*C[8]);
  G[3] = d*(C[1]*C[1]+C[4]*C[4]+C[7]*C[7]);
  G[4] = d*(C[1]*C[2]+C[4]*C[5]+C[7]*C[8]);
  G[5] = d*(C[2]*C[2]+C[5]*C[5]+C[8]*C[8]);

  // Compute 4-by-4 element matrices
  A[0]=G[0]; A[1]=G[1]; A[ 2]=G[2];
  A[4]=G[1]; A[5]=G[3]; A[ 6]=G[4];
  A[8]=G[2]; A[9]=G[4]; A[10]=G[5];
  A[ 3]=A[12]=-G[0]-G[1]-G[2];
  A[ 7]=A[13]=-G[1]-G[3]-G[4];
  A[11]=A[14]=-G[2]-G[4]-G[5];
  A[15]=G[0]+2*G[1]+2*G[2]+G[3]+2*G[4]+G[5];
}
```

Algorithm 7. Computing element matrices using cross-element vectorisation for the Laplacian with test and trial spaces of first-order Lagrange elements on tetrahedra. Vectorisation is implemented manually using AVX512 intrinsics and appropriate data types, such as `__m512d`.

have $\delta_{i,\mu(k)} \neq 0$ only if $f_i \subseteq T$. Therefore, the above sum can be restricted to those cells that contain the face f_i, so that

$$A_{i,j} = \sum_{\substack{T \in \mathcal{T} \\ f_i \subseteq T}} \left(A_T Q_T^{\mathsf{T}}\right)_{\eta(i),j},$$

where $\eta\colon [0, M) \to [0, m)$ is a left inverse of μ. That is, η maps global basis functions of the test space to basis functions of the local space $V(T)$.

Ultimately, the above may be translated to the high-level pseudo-code for a rowwise assembly algorithm that is shown in Algorithm 8.

The basic building blocks of the above algorithm are the same as those used in cellwise assembly, but there are some differences. Rather than computing and scattering the entire element matrix in steps 3 and 4, only a single row is needed in each iteration of the inner loop. Therefore, the order in which the element matrices are computed and scattered is different compared to a cellwise assembly, but the total amount of work carried out during these two steps remains the same. On the other hand, the first two steps, gathering a cell's vertex coordinates and transforming to a reference cell,

for all $i = 0, 1, \ldots, \dim V(\mathcal{T}) - 1$ **do**
  for all cells $T \in \mathcal{T}$ such that $f_i \subseteq T$ **do**
    1. Gather vertex coordinates of T
    2. Transform to a reference cell $\hat{T}$
    3. Compute row $\eta(i)$ of element matrix A_T
    4. Scatter row $\eta(i)$ of A_T to the i-th row of A
  end for
end for

Algorithm 8. Rowwise finite element assembly.

are now carried out more often than before. For example, for a tetrahedral mesh and first-order Lagrange elements, these steps occur exactly four times more often than in a cellwise assembly. That is, each cell consists of four vertices and steps 1 and 2 are executed every time a cell contains the vertex corresponding to the current row of the global matrix.

If we compare the cellwise and rowwise assembly algorithms, then the latter performs more work. However, the memory access pattern of the rowwise assembly is more regular because rows of the global matrix are accessed in sequence, and only a single row is written to during an iteration of the outer loop. This is in stark contrast to the irregular memory access pattern of cellwise assembly, which jumps back and forth between rows of the global matrix that are potentially far apart. Thus, the rowwise assembly algorithm presents an interesting trade-off by needing some redundant computations as well as extra memory traffic to gather vertex coordinates, but at the same time generating less memory traffic to write the results to the global matrix. More importantly, the rowwise algorithm can easily be parallelised without introducing any synchronisation overhead to avoid race conditions. This is achieved by simply assigning different rows of the global matrix to different threads.

5 Memory traffic estimates

In this section, we derive some lower and upper bounds of the memory traffic that are expected for the finite element assembly kernels from Section 3. These estimates are useful for evaluating the impact of memory traffic on different kernels and the effectiveness of some of the optimisations from Section 4.

We are concerned with multi-core CPU systems with one or more sockets that each contains multiple cores. Each CPU core is equipped with one or more floating-point functional units (FPUs) that perform floating-point calculations on operands that are supplied from the CPU core's working memory, or registers. Data that is needed for computation, but is not already located in registers, must be fetched from memory. Conversely, intermediate results that do not fit in registers, or results that are otherwise to be saved for later, must be written to memory. Moreover, memory is organised in a multi-level hierarchy, as depicted in Fig. 1. At the highest level of a typical memory hierarchy, that is, closest to the CPU registers, each core has a smaller, first-level (L1) cache and a larger second-level (L2) cache. Next, a third-level (L3) cache is usually

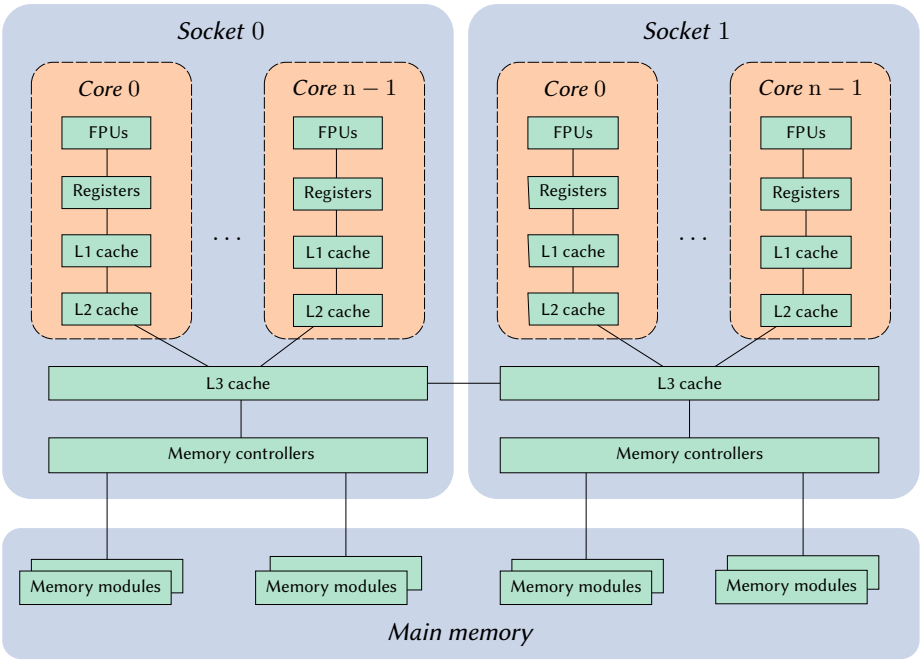


Figure 1. A simplified diagram of the architecture of a two-socket, multi-core CPU system and its memory hierarchy. Each core has one or more floating-point units (FPUs), a working memory, also called registers, and two levels of caches. A third-level cache is most often shared by all the cores of a socket. Finally, each socket has one or more memory controllers that are used to transfer data between third-level caches and main memory.

shared by some or all of the cores on a given socket. Finally, main memory, or DRAM, occupies the lowest level of the memory hierarchy. As a general rule, higher-level memories are faster, but have smaller storage capacities than lower-level ones.

5.1 Ideal cache model

For modelling purposes, it is often convenient to assume the *ideal cache model* [Frigo et al. 2012]. In this model, a cache is characterised by two properties. First, the cache size, Z, is the number of words (numerical values) that can be held in the cache at any time. Second, the main memory is partitioned into equally sized blocks, called cache blocks. The size of these blocks is referred to as the cache line size, L, the number of words held per block. The effect is that entire cache blocks are transferred, rather than individual values, whenever data is moved between cache and main memory. This allows exploiting spatial locality by presuming that nearby memory locations are likely to soon be accessed or written to.

The cache model also determines when and where cache blocks are placed in a cache. First, caches are assumed to operate with demand caching, which means that a cache block is placed into a cache only if a CPU issues a load or store that references

a memory location within the cache block. Second, we assume that caches are fully associative, which entails that a cache block can be placed anywhere in a cache. That is, unlike set associative or direct-mapped caches, there are no restrictions that make only one or a few cache lines available for placing a given cache block. As a consequence, the ideal cache model ignores conflict misses, or cache misses that would occur due to cache blocks being mapped to the same set of cache lines in a set-associative or direct-mapped cache. Third, if the cache is full, then a cache line must be evicted, and its contents written back to main memory, to make room for an incoming cache block. In this case, the ideal cache model employs an optimal replacement policy, which evicts cache lines in a way that minimises the number of transfers between the cache and main memory.

5.2 Gathering vertex coordinates

To gather the vertex coordinates of each mesh cell, the data must be brought from whichever level of the memory hierarchy where it resides to the processor's registers. In the following, the cache line size L denotes the number of double precision floating-point values that fit in a cache block, and we assume that $2L$ integers fit in the same cache block. The most common cache line size is 64 bytes, such that $L = 8$. Furthermore, let w denote the vector width, which is the number of consecutive double precision floating-point values that may be read or written with a single load or store. Similarly, we assume that twice the vector width, $2w$, applies when working with integers. In practice, the most relevant cases are $w = 1, 2, 4$ or 8.

First, note that the vertex indices are accessed sequentially. Unless the data resides in cache before the computation begins, there is no reuse of cached values. Therefore, in total $N \times \lceil m/(2w) \rceil$ loads are issued, where N is number of mesh cells and m is the number of vertices in each cell. Moreover, $\lceil Nm/(2L) \rceil$ cache blocks are transferred from lower levels of the memory hierarchy.

Second, reading the coordinates of a single vertex requires at least $\lceil d/w \rceil$ loads from the first-level cache, where d is the number of coordinates for each vertex. If each vertex coordinate is brought to the cache exactly once due to perfect cache data reuse, then the total number of cache lines that are transferred is at least $\lceil Md/L \rceil$, where M is the number of vertices in the mesh.

In the worst case, the coordinates of a vertex must be brought to the cache for every mesh cell that the vertex belongs to. If those coordinates are aligned to a cache line boundary, then they span $\lceil d/L \rceil$ cache lines. However, the vertex coordinates are usually not aligned to a cache line boundary, for instance, if $d = 3$ for a tetrahedral mesh and cache lines may hold $L = 8$ values. In this case, the coordinates of a vertex may span $\lceil d/L + 1 \rceil$ cache lines. Thus, the number of cache lines that are brought to the cache is at most $Nm \times \lceil d/L + 1 \rceil$.

In summary, $N \times \left(\lceil m/(2w) \rceil + m \lceil d/w \rceil \right)$ loads are issued, and the number of cache misses is at least $\lceil Nm/(2L) \rceil + \lceil Md/L \rceil$ and at most $\lceil Nm/(2L) \rceil + Nm \lceil d/L + 1 \rceil$.

5.3 Scattering results to a global matrix

Recall that adding an element matrix value to a global matrix first requires a binary search or a table lookup before the relevant matrix value is updated. In either case, updating the global matrix values results in an irregular memory access pattern that is influenced by the order of the mesh cells as well as the numbering of the global basis functions. To scatter the values of an $m \times n$ element matrix means accessing m different rows of the global matrix, and updating n values in each row. Since the matrix values that need to be updated usually do not occupy contiguous memory locations, vectorisation is of little use. Therefore, the global matrix updates amount to $N \times mn$ loads and the same number of stores, where N is the number of mesh cells. In the ideal case, where each global matrix value is brought to a cache exactly once, the number of cache blocks that must be transferred is at least $\lceil K/L \rceil$, where K is the number of nonzero entries in the global matrix.

If a binary-search approach is used, then there are $2Nm$ loads issued to read row pointers and an average of $Nmn \times \log_2(\bar{r})$ loads issued for reading column indices, where $\bar{r} = K/M$ is the average number of non-zeros per row and M is the number of matrix rows. The row pointers and column indices also result in at least $\lceil (M+1)/(2L) \rceil$ and $\lceil K/(2L) \rceil$ additional cache blocks being transferred, respectively.

If a table lookup is used, as in Algorithm 6, then the row pointers and column indices are no longer needed. Instead, there are $N \times mn$ loads issued and $\lceil Nmn/(2L) \rceil$ cache blocks transferred for the lookup table. Despite the extra memory usage due to the lookup table, this approach results in fewer memory accesses overall and the accesses to the lookup table are regular and in sequence.

To summarise, $N \times mn$ stores are issued regardless. In addition, the binary-search approach requires $2Nm + Nmn\big(1 + \log_2(\bar{r})\big)$ loads, whereas the lookup-table approach requires $2Nmn$ loads. Also, the binary-search approach causes at least $\lceil K/L \rceil + \lceil (M+1)/(2L) \rceil + \lceil K/(2L) \rceil$ cache misses, whereas the lookup-table counterpart incurs at least $\lceil K/L \rceil + \lceil Nmn/(2L) \rceil$ cache misses.

6 Numerical experiments

In this section, we describe numerical experiments that benchmark the performance of our finite element assembly algorithms. To understand the observed performance, we use measurements of memory traffic based on the hardware performance monitoring features of the CPUs and compare these with the memory traffic estimates from Section 5.

6.1 Experimental setup

The following is a brief description of the hardware and parameters used in the subsequent numerical experiments.

Table 1. Information about the hardware used in our experiments, including peak double-precision floating-point performance, cache sizes, memory configuration, and memory bandwidth. The practically achievable memory bandwidth is measured by the Triad kernel of the STREAM benchmark [McCalpin 2013].

	Intel Xeon Gold 6130	AMD Epyc 7601	Cavium TX2 CN9980
Instruction set	x86-64	x86-64	ARMv8.1
Microarchitecture	Skylake (server)	Zen	Vulcan
Cores per socket	16	32	32
Frequency	1.9 to 3.6 GHz	2.7 to 3.2 GHz	2.0 to 2.5 GHz
Double-precision floating-point perf.			
single core	112 Gflop/s	25.6 Gflop/s	20 Gflop/s
single socket	972.8 Gflop/s	691.2 Gflop/s	512 Gflop/s
Memory per socket			
L3 cache	22 MiB	64 MiB	32 MiB
DRAM channels	6	8	8
DRAM transfer rate	2.67 GHz	2.67 GHz	2.67 GHz
Theoretical max. bandwidth	128 GB/s	171 GB/s	171 GB/s
STREAM Triad bandwidth			
single core	13.0 GB/s	18.6 GB/s	11.7 GB/s
single socket	74.4 GB/s	81.3 GB/s	111.3 GB/s
dual socket	147.1 GB/s	161.4 GB/s	221.3 GB/s

6.1.1 Hardware

Most of the following experiments were conducted on a multi-core CPU system with two Intel Xeon Gold 6130 CPUs. In addition, we have used a dual-socket AMD Epyc 7601 system and a dual-socket Cavium ThunderX2 CN9980 system to study parallel performance and scalability in Section 6.3. See Table 1 for a summary of the main characteristics of these systems. For further details on the CPUs, see [Guo 2019; Intel Corporation 2018; Schor 2018].

The Xeon Gold 6130 CPU is based on the Skylake server microarchitecture, and therefore it supports the AVX512 instruction set, which is capable of performing 32 double precision floating-point operations per cycle if vectorisation and fused multiply-add instructions are used. If a single core is used, the CPU frequency is 3.6 GHz for scalar code and 3.5 GHz for AVX512, whereas if all sixteen cores are being used, the frequency is reduced to 2.8 GHz and 1.9 GHz for scalar and AVX512 operations, respectively. In comparison, the AMD Epyc and Cavium TX2 CPUs can perform only eight double-precision floating-point operations per cycle. Despite having twice as many CPU cores and slightly higher clock speeds, the peak floating-point performance of these CPUs is significantly lower than Intel Xeon.

Meanwhile, Intel Xeon has six memory channels, whereas AMD Epyc and Cavium TX2 have eight, which results in a higher memory bandwidth for the latter two systems. From the numbers measured by the Triad kernel of the STREAM bandwidth benchmark,

Table 2. Computational meshes used in our numerical experiments.

Mesh	Vertices	Cells
Uniform mesh 1	1 771 561	10 368 000
Uniform mesh 2	4 173 281	24 576 000
Cardiac mesh 1	1 255 775	6 735 654
Cardiac mesh 20	3 019 809	16 907 270
Cardiac mesh 41	2 226 802	12 255 517
Cardiac mesh 44	1 958 816	10 697 116
Aneurysm mesh 3	855 668	5 173 053
Aneurysm mesh 4	1 923 234	11 717 169

the realistic dual-socket memory bandwidth for the AMD Epyc system is about 10 % higher than Intel Xeon, while the dual-socket memory bandwidth for Cavium TX2 is almost 40 % higher than AMD Epyc.

The various benchmarks were compiled using GCC 9.2.0 with the options `-O3`, `-march=native`, and `-fopenmp`.

6.1.2 Computational meshes

In the following experiments, we use tetrahedral meshes from two different biomedical applications. The first set of meshes have previously been used in cardiac modelling [Marciniak et al. 2017], and they originate from patient data in a Danish study on cardiac disease [Jabbari et al. 2015]. The second set of meshes, which represent blood vessels with aneurysms, have been used in blood flow simulations, and they are based on data from the Aneurisk project [Aneurisk-Team 2012]. Finally, we also use two standard, uniform meshes of the unit cube. See Table 2 for an overview of the meshes.

6.2 Benchmarking assembly kernels

In this section, we report the results of benchmarking the individual assembly kernels from Section 3.

6.2.1 Gathering vertex coordinates

We begin by benchmarking the first assembly kernel, which is gathering the vertex coordinates of a mesh. Table 3 shows the observed performance for each mesh, both before and after the mesh has been reordered, as described in Section 4.1.

The results show that reordering yields a speedup for the cellwise algorithm of about 10x to 11x for the cardiac meshes and about 4x to 5x for the aneurysm meshes. Besides the performance, we have also measured the memory traffic by using the hardware performance events `skx_unc_imc[0-5]::UNC_M_CAS_COUNT.RD` that are associated with the CPU's memory controllers. These measurements show that the memory traffic for some of the original meshes is near the worst-case estimates that

Table 3. Performance and memory traffic for gathering vertex coordinates of tetrahedral meshes on Intel Xeon Gold 6130. The best- and worst case columns show best- and worst case estimates of the memory traffic, as described in Section 5.2.

	Perf. [Mcell/s]		DRAM read [MB]				Page walks [M]	
Mesh	Orig.	Reorder	Orig.	Reorder	Best case	Worst case	Orig.	Reorder
Uniform mesh 1	292	N/A	228	N/A	208	3483	0.0	N/A
Uniform mesh 2	292	N/A	537	N/A	493	8257	0.0	N/A
Cardiac mesh 1	27	271	1145	145	137	2263	18.3	0.0
Cardiac mesh 20	22	260	4232	378	342	5680	54.9	0.0
Cardiac mesh 41	23	260	2815	273	249	4117	38.1	0.0
Cardiac mesh 44	24	251	2330	235	218	3594	32.5	0.0
Aneurysm mesh 3	63	281	174	106	103	1738	2.6	0.0
Aneurysm mesh 4	49	268	764	248	233	3936	8.3	0.0

were derived in Section 5.2, whereas the reordered meshes yield memory traffic volumes close to the best-case estimates. In fact, for cellwise assembly, the original cardiac meshes generate about 8 to 10 times more memory traffic than the reordered versions, and about 1.7 to 3 times more for the aneurysm meshes.

In addition, the random access patterns associated with the original meshes trigger a large number of page walks, as measured by the hardware performance event `DTLB_LOAD_MISSES.WALK_COMPLETED`. These page walks occur due to missing the translation lookaside buffers, or TLBs, which are used to cache page table entries for translating virtual to physical memory addresses. Not only do page walks trigger additional memory accesses to fetch page table information from memory, but they also increase latency and reduce throughput for those memory accesses that miss the TLBs because they cannot complete before the page table information has been fetched. However, page walks are more or less eliminated by reordering the mesh, which in part explains the improved performance.

After reordering the mesh, we would expect the execution time to be limited by the available memory bandwidth. In practice, the achieved throughput is almost half of the 13.0 GB/s single-core bandwidth that is measured by STREAM and reported in Table 1. It is quite plausible that hardware prefetchers are less effective when faced with irregular memory access patterns rather than the sequential memory accesses performed by STREAM. Regardless, we cannot expect to attain much better performance for the reordered meshes, since the actual volumes of memory traffic are already close to the best-case estimates and cannot be significantly improved upon.

6.2.2 Computing element matrices

We have designed the benchmark for computing element matrices to be independent of the mesh connectivity and memory traffic concerns, by ensuring that data is only read from and written to the first-level cache. However, it does depend on the variational form that is to be assembled. In the following, we benchmark a kernel for computing the transformation to a reference mesh cell and two kernels for computing element

Table 4. Performance (in Mcell/s) for computing transformation to a reference cell and element matrices for the Laplacian on Intel Xeon Gold 6130. The "Auto" column relies on the compiler to perform automatic vectorisation, whereas the "Manual" column is for manually optimised code using AVX512 to perform cross-element vectorisation. Finally, the "Best case" is an upper bound on performance based on peak single core floating-point performance and a minimum number of floating-point operations per mesh cell.

	Auto	Manual	Best case
Transform to reference cell	205	1303	2732
$\mathcal{P}_1$ Laplacian	125	672	1750
$\mathcal{P}_2$ Laplacian	25	168	299

matrices for the Laplacian, $a(v, u) = \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x$. The performance of these kernels is presented in Table 4, including default kernels that rely on compiler autovectorisation and AVX512 kernels that use manually implemented cross-element vectorisation.

For the default kernels, we are relying on the compiler to automatically perform vectorisation, if possible. However, inspecting the generated assembly code reveals that it is unable to do so. Moreover, the performance ranges from about 7.6 to 14.4 Gflop/s. Observe that the lower number is close to the peak scalar floating-point performance of a single core when fused multiply-add instructions are not used, whereas 14.4 Gflop/s is precisely the scalar floating-point capacity of a single core when fused multiply-add instructions are used. Thus, the performance varies somewhat between the kernels, depending on whether or not the compiler is able to employ fused multiply-add instructions.

We observe a significant increase in throughput for the manually implemented cross-element vectorisation that uses AVX512 to compute eight element matrices simultaneously. More specifically, the transformation to a reference cell and the two Laplacian kernels all experience a speedup of about 5 to 8 times. Also, manual vectorisation attains almost half of the peak floating-point performance for the transform to reference cell, and 38 % and 56 % of peak floating-point performance for the first- and second-order Laplacian, respectively. This should be considered a high level of utilisation, if we take into account that fused multiply-add instructions may only be relevant for a portion of the overall calculations.

6.2.3 Scattering results to a global matrix

The final assembly kernels involve updating a global matrix based on element matrices that are computed in the previous step. In this case, we expect the performance to be highly dependent on the mesh and finite elements that are used, and, in particular, on the numbering of both the mesh cells and the global degrees of freedom of the finite element space. The performance of Algorithms 5 and 6 is shown in Table 5, for the original and reordered meshes and for first- and second-order Lagrange elements.

The first thing to note is the tremendous performance improvement that results from using a lookup table to locate non-zeros in the global matrix, rather than relying

Table 5. Performance and memory traffic for scattering element matrices to a global matrix on Intel Xeon Gold 6130.

	Performance [Mcell/s]		DRAM read / write [GB]			Page walks [M]	
Mesh	Original	Reordered	Original	Reordered	Best case	Original	Reordered
$\mathcal{P}_1$, cellwise, binary search							
Uniform mesh 1	19.8		0.58 / 0.27		0.49 / 0.21	0.0	
Uniform mesh 2	16.9		1.77 / 0.88		1.15 / 0.50	0.0	
Cardiac mesh 1	1.4	3.6	9.26 / 3.48	0.40 / 0.17	0.33 / 0.14	59.3	0.1
Cardiac mesh 20	1.2	3.6	25.56 / 8.87	1.15 / 0.50	0.81 / 0.35	164.4	0.3
Cardiac mesh 41	1.3	3.6	18.04 / 6.45	0.81 / 0.35	0.59 / 0.26	116.0	0.2
Cardiac mesh 44	1.3	3.6	15.36 / 5.53	0.67 / 0.29	0.51 / 0.22	99.9	0.2
Aneurysm mesh 3	2.5	3.6	3.16 / 1.38	0.29 / 0.12	0.24 / 0.10	14.8	0.1
Aneurysm mesh 4	2.3	3.7	8.14 / 3.39	0.71 / 0.31	0.55 / 0.24	35.7	0.2
$\mathcal{P}_1$, cellwise, lookup table							
Uniform mesh 1	99.8		0.93 / 0.26		0.87 / 0.21	0.0	
Uniform mesh 2	98.5		2.57 / 0.99		2.07 / 0.50	0.0	
Cardiac mesh 1	8.1	87.2	3.69 / 3.07	0.60 / 0.16	0.57 / 0.14	17.1	0.0
Cardiac mesh 20	7.2	79.6	10.05 / 8.45	1.62 / 0.48	1.43 / 0.35	46.6	0.1
Cardiac mesh 41	7.4	80.5	7.13 / 5.97	1.17 / 0.34	1.04 / 0.26	33.1	0.0
Cardiac mesh 44	7.5	84.8	6.14 / 5.15	0.97 / 0.26	0.91 / 0.22	28.6	0.0
Aneurysm mesh 3	15.8	87.7	1.43 / 1.03	0.46 / 0.12	0.44 / 0.10	4.2	0.0
Aneurysm mesh 4	13.9	83.7	3.78 / 2.84	1.07 / 0.29	0.98 / 0.24	10.0	0.0
$\mathcal{P}_1$, rowwise, lookup table							
Uniform mesh 1	83.2		1.05 / 0.21		1.05 / 0.21	0.0	
Uniform mesh 2	82.9		2.49 / 0.50		2.50 / 0.50	0.0	
Cardiac mesh 1	76.8	76.8	0.69 / 0.14	0.69 / 0.15	0.69 / 0.14	0.0	0.0
Cardiac mesh 20	77.6	76.5	1.72 / 0.35	1.72 / 0.35	1.73 / 0.35	0.1	0.1
Cardiac mesh 41	77.8	76.7	1.25 / 0.26	1.25 / 0.26	1.25 / 0.26	0.0	0.0
Cardiac mesh 44	76.4	76.0	1.09 / 0.23	1.09 / 0.22	1.10 / 0.22	0.0	0.0
Aneurysm mesh 3	79.7	77.7	0.52 / 0.10	0.52 / 0.10	0.52 / 0.10	0.0	0.0
Aneurysm mesh 4	79.4	76.8	1.18 / 0.24	1.19 / 0.24	1.19 / 0.24	0.0	0.0
$\mathcal{P}_2$, cellwise, lookup table							
Uniform mesh 1	6.7		10.79 / 5.16		7.34 / 3.20	0.6	
Uniform mesh 2	6.7		25.46 / 12.11		17.40 / 7.56	1.4	
Cardiac mesh 1	1.2	6.3	22.23 / 16.98	6.72 / 3.16	4.81 / 2.11	33.3	0.4
Cardiac mesh 20	1.1	6.1	57.45 / 43.24	17.92 / 8.48	12.02 / 5.26	85.5	1.3
Cardiac mesh 41	1.2	6.1	41.11 / 31.20	12.88 / 6.09	8.73 / 3.82	61.2	0.9
Cardiac mesh 44	1.3	6.2	35.67 / 27.15	11.07 / 5.23	7.62 / 3.34	53.2	0.8
Aneurysm mesh 3	1.6	6.3	14.01 / 9.70	5.03 / 2.31	3.66 / 1.59	13.7	0.4
Aneurysm mesh 4	1.7	6.1	31.65 / 21.90	12.03 / 5.61	8.29 / 3.60	30.1	0.9
$\mathcal{P}_2$, Rowwise, lookup table							
Uniform mesh 1	13.5		7.84 / 3.20		7.62 / 3.20	0.0	
Uniform mesh 2	13.5		18.57 / 7.58		18.05 / 7.56	0.0	
Cardiac mesh 1	12.6	12.6	5.14 / 2.12	5.13 / 2.12	4.99 / 2.11	0.1	0.1
Cardiac mesh 20	12.6	12.7	12.83 / 5.27	12.85 / 5.28	12.48 / 5.26	0.3	0.3
Cardiac mesh 41	12.6	12.7	9.31 / 3.83	9.32 / 3.83	9.06 / 3.82	0.2	0.3
Cardiac mesh 44	12.6	12.7	8.14 / 3.35	8.14 / 3.35	7.91 / 3.34	0.2	0.2
Aneurysm mesh 3	12.7	12.7	3.91 / 1.60	3.91 / 1.60	3.80 / 1.59	0.1	0.1
Aneurysm mesh 4	12.7	12.6	8.85 / 3.61	8.85 / 3.62	8.60 / 3.60	0.2	0.3

on a binary search. For first-order elements, the speedup is about 6x when the original mesh ordering is used and about 24x for the reordered meshes. The performance improvement for the original mesh ordering could partly be explained by the fact that the lookup table reduces memory traffic by half compared to the binary search. However, for the reordered meshes, it is in fact the lookup table that generates more memory traffic, in spite of the huge speedup that is observed. This indicates that the binary search is not at all limited by memory bandwidth in this case. More likely, the CPU is unable to ensure enough concurrent memory requests to use the memory subsystem effectively, perhaps due to conditional statements required in the search that could incur pipeline stalls. Recall also that the binary search issues significantly more loads, putting pressure on the connection between registers and the L1 cache, which may further impede accesses to lower levels of the memory hierarchy.

Second, for cellwise assembly with a lookup table, we observe that reordering the meshes results in a speedup of about 10x to 11x for the cardiac meshes and about 5x to 6x for the aneurysm meshes. This is closely correlated with a measured reduction in both read and write memory traffic. Overall, the reordering strategy works well, since the memory traffic ends up close to the best case, and there seems to be little room for improvement.

As expected, the rowwise algorithm is unaffected by mesh reordering. It achieves quite good performance regardless, as the cellwise algorithm is only about 5 % to 15 % faster in the case of first-order elements. This is consistent with the fact that the rowwise algorithm requires a small amount of additional data to be read for each mesh cell, namely the mapping from global degrees of freedom to the local degrees of freedom of each mesh cell. The fact that the rowwise algorithm results in memory traffic almost identical to the best case is strong evidence of its cache-friendly nature.

The results for second-order elements paint a similar picture. Mesh reordering results in a speedup of about 3x to 5x for cellwise assembly, whereas the performance of rowwise assembly is unaffected. However, in this case, the rowwise algorithm is almost twice as fast as the cellwise approach. The former also generates less read and write memory traffic, although the observed speedup seems larger than one would expect, if it were based only on the difference in memory traffic.

6.3 Parallel finite element assembly

Having considered each of the assembly kernels individually, we now turn our attention to the performance of the full cellwise and rowwise assembly algorithms. For the Intel Xeon Gold 6130 system, Table 6 compares the parallel performance of these algorithms with the reordered meshes when assembling the mass matrix and Laplacian for first- and second-order elements.

First, let us compare the serial performance of cellwise and rowwise assembly. Recall that the rowwise algorithm is required to gather cell vertex coordinates and compute the transformation to a reference cell several more times than the cellwise algorithm, as described in Section 4.5. This explains why the cellwise algorithm is 3 times faster when using first-order elements for both the mass matrix and Laplacian. However, for second-order elements, the difference in performance is much smaller. In this case, the rowwise algorithm is about 5 % to 10 % faster when assembling the mass

Table 6. Parallel performance (in Mcell/s) of cellwise and rowwise finite element assembly for the mass matrix and Laplacian on Intel Xeon Gold 6130.

	Cellwise performance [Mcell/s]						Rowwise performance [Mcell/s]					
	Number of threads						Number of threads					
Mesh	1	2	4	8	16	32	1	2	4	8	16	32
$\mathcal{P}_1$ Mass matrix												
Uniform mesh 1	75.4	18.9	36.6	67.8	119.6	238.0	29.5	59.3	113.8	211.0	363.4	550.8
Uniform mesh 2	74.5	19.0	36.6	67.6	119.4	237.8	31.2	59.1	102.4	209.1	363.2	544.5
Cardiac mesh 1	56.1	18.4	35.2	65.2	115.8	218.4	21.0	38.9	73.1	133.8	238.3	437.3
Cardiac mesh 20	58.6	18.1	35.0	64.7	114.9	212.1	20.2	37.0	71.1	131.8	233.3	427.1
Cardiac mesh 41	59.7	18.2	35.0	64.8	115.3	215.1	20.3	37.9	72.1	131.7	235.8	441.1
Cardiac mesh 44	62.0	18.2	35.1	65.0	115.4	214.9	20.6	37.3	71.2	131.5	236.8	442.4
Aneurysm mesh 3	64.6	18.4	35.2	64.7	115.2	217.2	22.1	41.6	76.8	140.8	246.7	474.3
Aneurysm mesh 4	61.5	18.3	34.8	64.4	114.8	210.0	21.8	40.0	73.4	136.9	238.5	466.9
$\mathcal{P}_1$ Laplacian												
Uniform mesh 1	50.2	18.1	35.0	64.7	114.7	221.7	18.2	33.7	66.6	122.9	218.1	435.9
Uniform mesh 2	49.7	17.9	34.9	64.7	114.1	222.3	18.2	34.5	66.4	122.0	217.3	430.0
Cardiac mesh 1	44.4	17.6	33.8	62.5	111.5	207.5	14.8	27.6	52.3	96.1	170.3	337.0
Cardiac mesh 20	41.1	17.4	33.5	61.9	110.1	200.7	14.4	26.4	50.8	94.0	167.4	324.7
Cardiac mesh 41	41.8	17.5	33.6	62.1	110.7	202.2	14.4	26.9	51.5	93.8	169.0	330.8
Cardiac mesh 44	42.9	17.5	33.6	62.2	110.8	203.0	14.5	26.5	50.5	93.5	168.1	333.4
Aneurysm mesh 3	44.7	17.6	33.8	62.4	108.3	204.7	15.3	28.7	53.5	98.7	175.2	353.0
Aneurysm mesh 4	42.7	17.5	33.4	61.8	110.2	197.5	15.0	27.9	51.6	95.6	169.6	339.3
$\mathcal{P}_2$ Mass matrix												
Uniform mesh 1	5.7	3.0	5.8	10.6	19.2	25.8	7.4	11.2	15.3	18.4	29.1	38.2
Uniform mesh 2	5.6	3.0	5.8	10.6	18.9	24.7	7.4	11.3	15.2	18.4	29.1	38.5
Cardiac mesh 1	5.4	2.9	5.6	10.4	18.6	24.6	5.5	8.5	11.9	15.2	25.4	34.6
Cardiac mesh 20	5.2	2.9	5.6	10.4	18.5	24.1	5.4	8.3	11.4	14.6	24.1	33.5
Cardiac mesh 41	5.2	2.9	5.6	10.3	18.3	24.0	5.5	8.4	11.3	14.9	24.9	34.4
Cardiac mesh 44	5.2	2.9	5.6	10.3	18.5	24.1	5.5	8.4	11.7	15.0	24.7	34.4
Aneurysm mesh 3	5.3	2.9	5.6	10.3	18.5	23.9	5.8	8.8	12.3	14.9	24.9	34.1
Aneurysm mesh 4	5.2	2.9	5.6	10.3	18.4	23.6	5.8	8.7	12.0	14.8	24.6	34.2
$\mathcal{P}_2$ Laplacian												
Uniform mesh 1	5.4	2.9	5.6	10.1	18.4	25.6	4.4	6.8	9.6	12.1	21.7	31.4
Uniform mesh 2	5.3	2.9	5.6	10.1	18.2	24.5	4.4	6.8	9.6	12.1	21.7	31.5
Cardiac mesh 1	5.1	2.8	5.4	10.0	17.9	24.5	3.8	6.0	8.4	10.9	19.7	29.3
Cardiac mesh 20	4.9	2.8	5.4	9.9	17.8	24.0	3.8	5.8	8.2	10.6	18.8	28.4
Cardiac mesh 41	4.9	2.8	5.4	9.9	17.8	23.9	3.8	5.9	8.2	10.7	19.2	29.3
Cardiac mesh 44	5.0	2.8	5.4	9.9	17.8	23.9	3.8	5.9	8.2	10.8	18.9	29.3
Aneurysm mesh 3	5.1	2.8	5.4	9.9	17.7	23.7	3.9	6.0	8.5	10.5	19.0	28.4
Aneurysm mesh 4	4.9	2.8	5.4	9.9	17.7	23.5	3.9	6.0	8.4	10.4	18.4	28.6

matrix, but the cellwise algorithm is about 30 % faster at assembling the Laplacian. Because of the much larger element matrices associated with second-order elements, it seems clear that the scattering kernel is more dominant than it is for first-order elements. Thus, the rowwise assembly algorithm suffers less from the additional work it must do in connection with the first two kernels. In any case, the cellwise assembly algorithm is preferable for a sequential assembly, particularly for first-order elements.

As a way of validating the performance results for the sequential, cellwise assembly, we have also compared with the performance of the open source finite element code in FEniCS [Logg et al. 2012]. Note that FEniCS uses a cellwise assembly algorithm with the binary search strategy to locate global matrix entries after each element matrix has been computed. To make a fair comparison, we ran our benchmark for a cellwise assembly algorithm to assemble the Laplacian for first-order Lagrange elements, also using the binary search strategy. The performance of our benchmark for the reordered version of "Cardiac mesh 20" is 3.53 Mcell/s, whereas the performance of FEniCS is 2.37 Mcell/s with the same mesh. The performance of the two implementations is not too different, especially considering that FEniCS is a much more general code that is designed to handle more advanced use cases than our simple benchmark.

Regarding parallel performance, both the cellwise and rowwise algorithms experience some slowdown at first compared to the sequential cellwise assembly. We already know that the rowwise algorithm is about three times slower than cellwise assembly when using a single thread, but we now observe that rowwise assembly is faster whenever four or more threads are used. Furthermore, the algorithm scales nicely as the number of threads increases, and the performance reaches about 300 to 550 Mcell/s and 28 to 38 Mcell/s for first- and second-order elements, respectively, when 32 threads are used. The parallel performance of the cellwise assembly only reaches 200 to 240 Mcell/s for first-order elements and 23 to 26 Mcell/s for second-order elements. This is because the parallel cellwise assembly seems to suffer from the high cost, or synchronisation overhead, that is introduced through the use of atomics to avoid race conditions.

To provide a more complete picture, Table 7 shows the parallel performance of a rowwise assembly for the Laplacian with first-order elements on three different multi-core systems, including Intel Xeon Gold 6130, AMD Epyc 7601 and Cavium TX2 CN9980. For the latter two systems, the serial performance lags behind Intel Xeon. This is partly due to the CPU cores operating at a lower frequency than the Intel Xeon, although the Cavium TX2 also seems less able to make use of instruction-level parallelism. The rowwise assembly reaches about 340 to 400 Mcell/s and 310 to 380 Mcell/s on AMD Epyc and Cavium TX2, respectively.

A visual comparison of the performance of the three systems is shown in Fig. 2 for "Cardiac mesh 20". In the whole-system dual-socket case, AMD Epyc is faster than both Intel Xeon and Cavium TX2. The performance of the latter two systems is comparable, in spite of Cavium TX2 having twice as many cores and a significantly higher memory bandwidth. Thus, the finite element assembly performance is contrary to the STREAM Triad results in Table 1, which show that Cavium TX2 has the highest single- and dual-socket memory bandwidth of the three systems. This leads us to conclude that performance on Cavium TX2 is not primarily limited by the available memory bandwidth.

Table 7. Parallel performance (in Mcell/s) of rowwise finite element assembly for the Laplacian with first-order Lagrange elements on Intel Xeon Gold 6130, AMD Epyc 7601, and Cavium TX2 CN9980.

	Rowwise performance [Mcell/s]						
	Number of threads						
Mesh	1	2	4	8	16	32	64
Intel Xeon Gold 6130							
Uniform mesh 1	18.2	33.7	66.6	122.9	218.1	435.9	
Uniform mesh 2	18.2	34.5	66.4	122.0	217.3	430.0	
Cardiac mesh 1	14.8	27.6	52.3	96.1	170.3	337.0	
Cardiac mesh 20	14.4	26.4	50.8	94.0	167.4	324.7	
Cardiac mesh 41	14.4	26.9	51.5	93.8	169.0	330.8	
Cardiac mesh 44	14.5	26.5	50.5	93.5	168.1	333.4	
Aneurysm mesh 3	15.3	28.7	53.5	98.7	175.2	353.0	
Aneurysm mesh 4	15.0	27.9	51.6	95.6	169.6	339.3	
AMD Epyc 7601							
Uniform mesh 1	11.9	23.3	45.2	86.4	165.4	292.1	394.9
Uniform mesh 2	12.3	23.3	45.2	88.6	173.6	321.2	387.3
Cardiac mesh 1	11.2	20.4	37.9	72.2	133.3	246.2	383.4
Cardiac mesh 20	10.5	20.4	37.4	71.8	134.7	247.3	393.2
Cardiac mesh 41	10.6	19.7	37.4	71.5	135.6	255.4	383.1
Cardiac mesh 44	10.8	20.3	36.8	71.0	137.1	256.6	350.1
Aneurysm mesh 3	10.7	21.1	38.9	75.1	139.0	253.0	338.2
Aneurysm mesh 4	11.4	21.0	38.4	73.6	140.8	262.0	403.1
Cavium TX2 CN9980							
Uniform mesh 1	6.5	12.8	24.7	49.2	97.8	193.9	382.4
Uniform mesh 2	6.3	12.3	24.3	48.3	96.6	191.4	381.7
Cardiac mesh 1	6.1	12.1	23.4	45.4	87.8	171.9	340.3
Cardiac mesh 20	6.0	11.6	22.8	44.4	86.3	168.3	320.9
Cardiac mesh 41	6.0	11.9	23.0	43.8	87.1	167.7	320.9
Cardiac mesh 44	6.0	11.8	22.8	44.4	86.0	167.3	312.5
Aneurysm mesh 3	6.1	12.1	24.1	46.8	89.6	172.2	328.3
Aneurysm mesh 4	6.1	11.9	22.7	44.1	85.8	167.8	327.7

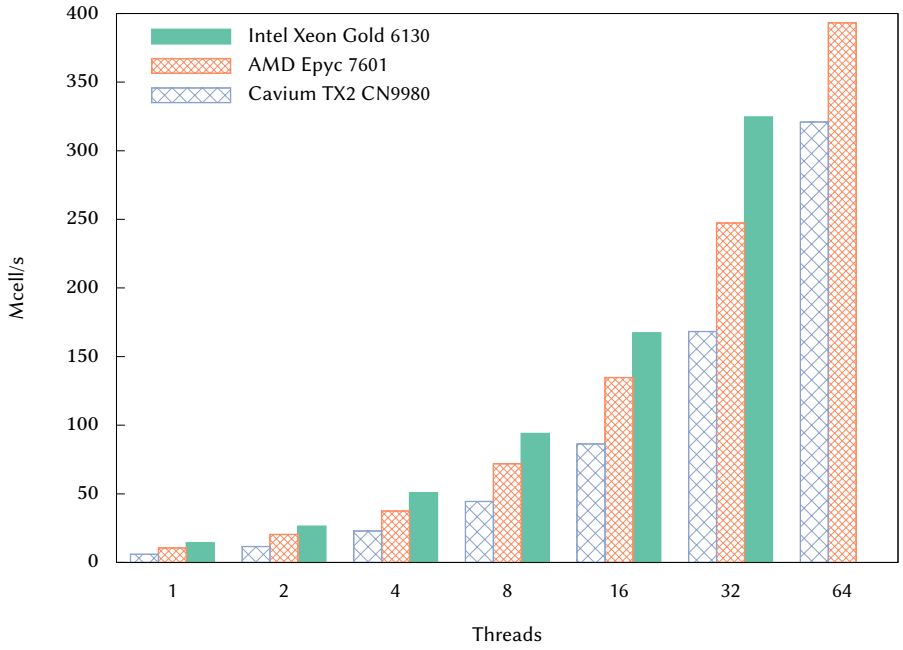


Figure 2. Parallel performance for rowwise assembly of the Laplacian with first-order Lagrange elements for “Cardiac mesh 20”.

7 Related work

The standard cellwise assembly algorithm is described, for example, by Ern and Guermond [2004], and in the FEniCS book [Logg et al. 2012]. The tensor representation for computing element matrices is described by Kirby et al. [2005] and Kirby and Logg [2006], and it was extended to more general finite elements and variational forms by Rognes et al. [2009]. Besides the tensor representation, the optimisation of kernels for computing element matrices has been studied from several different angles in connection with automated code generation for FEniCS [Logg et al. 2012] and Firedrake [Rathgeber et al. 2016]. This includes numerical integration [Ølgaard and Wells 2010], the use of computer algebra for exact integration and simplification of expressions [Alnæs and Mardal 2010; Russell and Kelly 2013], as well as low-level loop optimisations [Homolya et al. 2018; Luporini et al. 2017; Luporini et al. 2015].

A lot of work on optimising finite element assembly has focused on reducing the number of floating-point operations that are required to compute element matrices, for instance, using a technique called sum factorisation [Bolis et al. 2014; Cantwell et al. 2011; Kronbichler and Kormann 2012; Vos et al. 2010], which exploits tensor-product structure that is particularly relevant for quadrilateral and hexahedral meshes and higher-order polynomial spaces. Sometimes, a fully assembled global matrix itself is not needed, but it is sufficient to evaluate the product of such a matrix with a vector.

In these circumstances, it is possible to use matrix-free methods [Kronbichler and Kormann 2012] or variations, such as the local matrix approach [Cantwell et al. 2011; Markall et al. 2013; Vos et al. 2010].

Sun et al. [2020] describe the use of cross-element vectorisation on Intel multi-core CPUs using AVX-512, and demonstrate some significant performance improvements for more complicated variational forms. A similar idea is used by Knepley and Terrel [2013] in their implementation of a GPU-based element matrix computation.

More recently, a lot of work has focused on implementing finite element assembly algorithms for GPUs. Cecka et al. [2011] explored several strategies for global assembly, including both cellwise and rowwise algorithms. Markall et al. [2013] compare a parallel, global matrix assembly that uses a mesh colouring or atomics to avoid race conditions to what they call a local matrix approach. The latter is like a matrix-free method except that the element matrices are computed only once, and they are stored in memory and reused each time a matrix-vector multiplication is computed. Fu et al. [2014] implement a cellwise assembly algorithm that uses a binary search when scattering element matrices to a global matrix. Their method also relies on partitioning the mesh to ensure that data fits in a GPU's fast shared memory. Reguly and Giles [2015] perform a detailed comparison of both the assembly and solution of linear systems, where they compare cellwise, global matrix assembly for CSR and ELLPACK sparse matrix formats to the local matrix- and matrix-free methods. In addition to various experiments that demonstrate the tradeoffs between different methods, they also describe the memory traffic required by each method.

8 Conclusion

Finite element assembly algorithms with linear or quadratic finite elements on unstructured meshes are characterised by low arithmetic intensity and irregular memory access patterns. As a result, moving data to and from memory can greatly impact the performance. We have performed a detailed breakdown of the standard cellwise finite element assembly algorithm, focusing on a quantified understanding of the memory traffic involved. Estimated lower and upper bounds of the memory traffic have been derived and used to demonstrate the effectiveness of some optimisations related to, specifically, mesh reordering and use of a lookup table in the scattering kernel. In addition, we have shown that a rowwise assembly algorithm, which has previously been used for finite element assembly on GPUs, is also an efficient strategy for parallel assembly with shared memory on multi-core CPUs.

Acknowledgements

This work was supported by the Research Council of Norway under contract 251186. Also, the research presented in this paper has benefited from the Experimental Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053.

References

Alnæs, M. S. and K.-A. Mardal (Jan. 2010). "On the Efficiency of Symbolic Computations Combined with Code Generation for Finite Element Methods". In: *ACM Trans. Math. Softw.* 37.1. ISSN: 0098-3500. DOI: 10.1145/1644001.1644007.

Aneurisk-Team (June 2012). *AneuriskWeb project website*. Emory University, Department of Math&CS. URL: http://ecm2.mathcs.emory.edu/aneuriskweb.

Bolis, A., C. D. Cantwell, R. M. Kirby, and S. J. Sherwin (Apr. 2014). "From h to p efficiently: optimal implementation strategies for explicit time-dependent problems using the spectral/hp element method". In: *International Journal for Numerical Methods in Fluids* 75.8, pp. 591–607. ISSN: 0271-2091. DOI: 10.1002/fld.3909.

Cantwell, C. D., S. J. Sherwin, R. M. Kirby, and P. H. J. Kelly (Apr. 2011). "From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements". In: *Computers & Fluids* 43.1, pp. 23–28. ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2010.08.012.

Cecka, C., A. J. Lew, and E. Darve (Aug. 2011). "Assembly of finite element methods on graphics processors". In: *International Journal for Numerical Methods in Engineering* 85.5, pp. 640–669. DOI: 10.1002/nme.2989.

Ciarlet, P. G. (2002). *The Finite Element Method for Elliptic Problems*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics. ISBN: 0-89871-514-8.

Cuthill, E. and J. McKee (1969). "Reducing the Bandwidth of Sparse Symmetric Matrices". In: *Proceedings of the 1969 24th National Conference*. ACM '69. New York, NY, USA: Association for Computing Machinery, pp. 157–172. ISBN: 9781450374934. DOI: 10.1145/800195.805928.

Ern, A. and J.-L. Guermond (2004). *Theory and Practice of Finite Elements*. Applied Mathematical Sciences. Springer. ISBN: 0-387-20574-8.

Frigo, M., C. E. Leiserson, H. Prokop, and S. Ramachandran (Jan. 2012). "Cache-Oblivious Algorithms". In: *ACM Transactions on Algorithms* 8.1, 4:1–4:22. ISSN: 1549-6325. DOI: 10.1145/2071379.2071383.

Fu, Z., T. J. Lewis, R. M. Kirby, and R. T. Whitaker (Feb. 2014). "Architecting the Finite Element Method Pipeline for the GPU." In: *Journal of Computational and Applied Mathematics* 257, pp. 195–211. ISSN: 0377-0427. DOI: 10.1016/j.cam.2013.09.001.

George, A. (1973). "Nested Dissection of a Regular Finite Element Mesh". In: *SIAM Journal on Numerical Analysis* 10.2, pp. 345–363. ISSN: 0036-1429. DOI: 10.1137/0710032.

George, A. and D. R. Mcintyre (1978). "On the Application of the Minimum Degree Algorithm to Finite Element Systems". In: *SIAM Journal on Numerical Analysis* 15.1, pp. 90–112. ISSN: 0036-1429. DOI: 10.1007/BFb0064460.

Guo, X. (Feb. 2019). *Best Practice Guide - AMD EPYC*. Ed. by O. W. Saastad. Partnership for Advanced Computing in Europe (PRACE). URL: https://prace-ri.eu/wp-content/uploads/Best-Practice-Guide_AMD.pdf.

Homolya, M., L. Mitchell, F. Luporini, and D. A. Ham (June 2018). "TSFC: A Structure-Preserving Form Compiler". In: *SIAM Journal on Scientific Computing* 40.3, pp. 401–428. ISSN: 1064-8275. DOI: 10.1137/17M1130642.

Intel Corporation (Apr. 2018). *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 248966-040. Intel Corporation.

Jabbari, R., T. Engstrøm, C. Glinge, B. Risgaard, J. Jabbari, B. G. Winkel, C. J. Terkelsen, H.-H. Tilsted, L. O. Jensen, M. Hougaard, S. E. Chiuve, F. Pedersen, J. H. Svendsen, S. Haunsø, C. M. Albert, and J. Tfelt-Hansen (Jan. 2015). "Incidence and risk factors of ventricular fibrillation before primary angioplasty in patients with first ST-elevation myocardial infarction: a nationwide study in Denmark". In: *Journal of the American Heart Association* 4.1. ISSN: 2047-9980. DOI: 10.1161/JAHA.114.001399.

Kirby, R. C., M. Knepley, A. Logg, and L. R. Scott (2005). "Optimizing the Evaluation of Finite Element Matrices". In: *SIAM Journal on Scientific Computing* 27.3, pp. 741–758. ISSN: 1064-8275. DOI: 10.1137/040607824.

Kirby, R. C. and A. Logg (Sept. 2006). "A compiler for variational forms". In: *ACM Trans. Math. Softw.* 32.3, pp. 417–444. ISSN: 0098-3500. DOI: 10.1145/1163641.1163644.

Knepley, M. G. and A. R. Terrel (Feb. 2013). "Finite Element Integration on GPUs". In: *ACM Transactions on Mathematical Software* 39.2. ISSN: 0098-3500. DOI: 10.1145/2427023.2427027.

Kronbichler, M. and K. Kormann (June 2012). "A generic interface for parallel cell-based finite element operator application". In: *Computers and Fluids* 63, pp. 135–147. ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2012.04.012.

Logg, A., K.-A. Mardal, G. N. Wells, et al. (2012). *Automated Solution of Differential Equations by the Finite Element Method*. Berlin: Springer. ISBN: 978-3-642-23098-1. DOI: 10.1007/978-3-642-23099-8.

Luporini, F., D. A. Ham, and P. H. J. Kelly (Mar. 2017). "An Algorithm for the Optimization of Finite Element Integration Loops". In: *ACM Transactions on Mathematical Software* 44.1. ISSN: 0098-3500. DOI: 10.1145/3054944.

Luporini, F., A. L. Varbanescu, F. Rathgeber, G.-T. Bercea, J. Ramanujam, D. A. Ham, and P. H. J. Kelly (Jan. 2015). "Cross-Loop Optimization of Arithmetic Intensity for Finite Element Local Assembly". In: *ACM Transactions on Architecture and Code Optimization* 11.4. ISSN: 1544-3566. DOI: 10.1145/2687415.

Marciniak, M., H. Arevalo, J. Tfelt-Hansen, T. Jespersen, R. Jabbari, C. Glinge, K. A. Ahtarovski, N. Vejlstrup, T. Engstrom, M. M. Maleckar, and K. McLeod (Jan. 2017). "From CMR Image to Patient-Specific Simulation and Population-Based Analysis: Tutorial for an Openly Available Image-Processing Pipeline". In: *STACOM 2016: Statistical Atlases and Computational Models of the Heart. Imaging and Modelling Challenges*. Ed. by T. Mansi, K. McLeod, M. Pop, K. Rhode, M. Sermesant, and A. Young. Springer International Publishing, pp. 106–117. ISBN: 978-3-319-52718-5. DOI: 10.1007/978-3-319-52718-5_12.

Markall, G. R., A. Slemmer, D. A. Ham, P. H. J. Kelly, C. D. Cantwell, and S. J. Sherwin (Jan. 2013). "Finite element assembly strategies on multi-core and many-core architectures". In: *International Journal for Numerical Methods in Fluids* 71.1, pp. 80–97. ISSN: 0271-2091. DOI: 10.1002/fld.3648.

McCalpin, J. D. (Jan. 2013). *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Department of Computer Science School of Engineering and Applied Science, University of Virginia. Charlottesville, Virginia. URL: https://www.cs.virginia.edu/stream/.

Ølgaard, K. B. and G. N. Wells (Jan. 2010). "Optimizations for quadrature representations of finite element tensors through automated code generation". In: *ACM Transactions on Mathematical Software* 37.1. ISSN: 0098-3500. DOI: 10.1145/1644001.1644009.

Rathgeber, F., D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly (Dec. 2016). "Firedrake: Automating the Finite Element Method by Composing Abstractions". In: *ACM Transactions on Mathematical Software* 43.3. ISSN: 0098-3500. DOI: 10.1145/2998441.

Reguly, I. Z. and M. B. Giles (Apr. 2015). "Finite Element Algorithms and Data Structures on Graphical Processing Units". In: *International Journal of Parallel Programming* 43.2, pp. 203–239. ISSN: 0885-7458. DOI: 10.1007/s10766-013-0301-6.

Rognes, M. E., R. C. Kirby, and A. Logg (Nov. 2009). "Efficient Assembly of $H(\mathrm{div})$ and $H(\mathrm{curl})$ Conforming Finite Elements". In: *SIAM Journal on Scientific Computing* 31.6, pp. 4130–4151. ISSN: 1064-8275. DOI: 10.1137/08073901X.

Russell, F. P. and P. H. J. Kelly (July 2013). "Optimized Code Generation for Finite Element Local Assembly Using Symbolic Manipulation". In: *ACM Transactions on Mathematical Software* 39.4. ISSN: 0098-3500. DOI: 10.1145/2491491.2491496.

Schor, D. (June 2018). *A Look at Cavium's New High-Performance ARM Microprocessors and the Isambard Supercomputer*. WikiChip. URL: https://fuse.wikichip.org/news/1316/a-look-at-caviums-new-high-performance-arm-microprocessors-and-the-isambard-supercomputer/.

Sun, T., L. Mitchell, K. Kulkarni, A. Klöckner, D. A. Ham, and P. H. J. Kelly (May 2020). *A study of vectorization for matrix-free finite element methods*. arXiv: 1903.08243. URL: http://arxiv.org/abs/1903.08243.

Tchiboukdjian, M., V. Danjean, and B. Raffin (Jan. 2010). "Binary Mesh Partitioning for Cache-Efficient Visualization". In: *IEEE Transactions on Visualization and Computer Graphics* 16.5, pp. 815–828. ISSN: 2160-9306. DOI: 10.1109/TVCG.2010.19.

Tchiboukdjian, M., V. Danjean, and B. Raffin (2008). "A Fast Cache-Oblivious Mesh Layout with Theoretical Guarantees". In: *International Workshop on Super Visualization (IWSV'08)*. Kos, Greece. URL: https://hal.inria.fr/inria-00436053.

Vos, P. E., S. J. Sherwin, and R. M. Kirby (2010). "From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low- and high-order discretisations". In: *Journal of Computational Physics* 229.13, pp. 5161–5181. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2010.03.031.

Yoon, S.-E. and P. Lindstrom (Nov. 2006). "Mesh Layouts for Block-Based Caches". In: *IEEE Transactions on Visualization and Computer Graphics* 12.5, pp. 1213–1220. ISSN: 1077-2626. DOI: 10.1109/TVCG.2006.162.

Paper III

Leveraging GPU-accelerated finite element computation with automated code generation: A holistic approach

James D. Trotter, Simula Research Laboratory and University of Oslo, Norway
Johannes Langguth, Simula Research Laboratory
Xing Cai, Simula Research Laboratory and University of Oslo, Norway

Submitted for publication.

Abstract

Automated code generation has been increasingly used to enable complex finite element computations on parallel architectures. The FEniCS framework is an example of this trend, where user friendliness is provided by a mathematics-like Unified Form Language that allows users to easily prescribe partial differential equations (PDEs) and numerical details of the finite element method. The automated compiler inside FEniCS then translates the high-level user input into parallelised C++ code. While the capabilities and performance of auto-generated FEniCS code have been verified by real-world applications running on clusters of multi-core CPUs, GPU acceleration at the moment only applies to the stage that solves linear systems derived from the finite element method. The preceding phase of discretising PDEs by assembling those linear systems, is restricted to CPUs.

In this paper, we extend the FEniCS compiler to also generate code for GPU-based assembly of linear systems, including auto-generated optimisations of the resulting CUDA C++ code. Numerical experiments show that GPU-based linear system assembly for a typical PDE benefits from using a lookup table to avoid repeatedly carrying out numerous binary searches, and that further performance gains are realised by assembling a matrix row by row.

More importantly, the extended FEniCS compiler seamlessly couples the assembly and solution phases for GPU acceleration, so that we are able to eliminate all unnecessary CPU-GPU data transfers. Further experiments are used to quantify the negative impact of these data transfers, which can entirely destroy the potential of GPU acceleration if the assembly and solution phases are offloaded to GPU separately. Finally, a complete, auto-generated GPU-based PDE solver for a non-linear solid mechanics application is used to demonstrate a substantial speedup over running on dual-socket multi-core CPUs.

1 Introduction

Numerically solving partial differential equations (PDEs) is the most important problem in the area of scientific computing, and the finite element method (FEM) is one of the most important tools for doing so. Such computations are often implemented as hand-written, manually optimised computational kernels tailored to each individual problem. However, developing high-performance FEM codes is very challenging for domain scientists who are not experts in high-performance computing. Writing codes for accelerators such as GPUs is even more difficult, but the performance and energy efficiency of modern accelerators makes such codes very attractive.

Thus, an alternative approach is to offer a high-level language in which finite element-based computations may be expressed, and then automatically generate low-level, optimised code for problem-specific kernels. This strategy is used, for instance, by the open-source frameworks FEniCS [Logg et al. 2012] and Firedrake [Rathgeber et al. 2016] for parallel, distributed-memory, CPU-based PDE solvers. Among software packages and libraries that use FEM to solve PDEs, so far only a few offer GPU-accelerated computations. This is mostly due to the complexities of GPU programming associated with implementing the intricate details of finite element computations and the difficulty of obtaining performance gains that offset the overhead of offloading calculations to GPUs.

This paper describes how we have extended FEniCS with automated generation of GPU code for finite element assembly, thereby enabling fully GPU-based finite element solvers. Our contribution is not limited to providing a new and improved GPU acceleration to the widely used FEniCS framework. We also use automated code generation to achieve a seamless integration between the two important phases of finite element computations, thereby eliminating the need for most CPU-GPU transfers. To the best of our knowledge, no prior results exist in this generic topic. Moreover, we carefully study the impact of various CUDA optimisations applicable to auto-generated finite element code. Our work is fully compatible with all the other features of FEniCS, which means that non-linearity, vector PDEs, unstructured meshes, complicated boundary conditions, and so on, pose no hindrance to GPU acceleration.

The rest of the paper is organised as follows. First, we provide some background on automated code generation for FEM and the FEniCS PDE solver framework in Section 2, along with some basic principles of GPU computing. In Section 3, we focus on GPU-based finite element assembly and show how we accordingly extended the built-in automated code generation features of the FEniCS form compiler. Next, we discuss optimisations that are relevant to our GPU-based assembly algorithms in Section 4. In Section 5, we present numerical experiments and results to illustrate key parts of the development and optimisation process, including the effectiveness of various optimisations and the performance of the final GPU-based assembly algorithm. This is followed by a discussion of related work in Section 6 and concluding remarks in Section 7.

2 Finite element methods and automated code generation

The popularity of FEM is due to a solid mathematical foundation and the flexibility in handling irregular geometries, plus a wealth of shape functions to represent the numerical solutions of PDEs. The operation of finite element solvers consists of two major phases: *assembling a linear system* and *solving it*. Since solving a linear system is a generic kernel used in many types of numerical computations, its GPU acceleration is a well studied subject [Anzt et al. 2020]. On the other hand, the phase of linear system assembly, which discretises a PDE following the finite element principle, is a far less studied area with respect to automated code generation and, particularly, GPU acceleration.

2.1 Finite element assembly

By omitting many of the mathematical details, we will give here a very brief description of the finite element assembly phase. The purpose is to show the main computational steps involved, and explain how automated code generation and GPU acceleration can be applied.

The algorithmic procedure within the finite element assembly phase is generic, independent of the target PDE or the choice of shape functions. Here, the solution domain of the target PDE is represented by a mesh of non-overlapping cells, called *elements*. The numerical solution is sought in each element, combining prescribed shape functions (also called *trial* functions). There are many choices of the element type, such as triangles in 2D and tetrahedra in 3D. Given an element type, the associated shape functions are typically piecewise polynomials. The overall assembly procedure loops over each element T, computing an element matrix A_T and an element vector b_T by calculating integrals based on the so-called *variational form* of the target PDE (an example will be given below for Poisson's equation).

The global coefficient matrix A and right-hand side vector b of the resulting linear system, $Ax = b$, arise from assembling all the element matrices and vectors. Since the shape functions can span across neighbouring elements, each non-zero value in A or b is normally a sum of the corresponding values from several element matrices or vectors. (A needed local-to-global mapping is decided beforehand.) The global coefficient matrix A is always sparse, because each shape function is only non-zero in a small number of elements. If the computational mesh is unstructured, the non-zero values are irregularly placed in A. The solution vector x will contain the weights for combining the shape functions into the overall numerical solution of the target PDE.

Example: Poisson's equation

To provide a more concrete example, we consider Poisson's equation on a polygonal domain $\Omega \subset \mathbf{R}^d$ with a very simple boundary condition,

$$-\kappa\nabla^2 u = f \quad \text{in } \Omega, \qquad u = 0 \quad \text{on } \partial\Omega, \tag{1}$$

where $\kappa > 0$ is a constant and f is a given source term. The corresponding variational form has two parts (a bilinear form and a linear form) as follows:

$$a(v,u) = \int_\Omega \kappa \nabla v \cdot \nabla u \, \mathrm{d}x, \quad L(v) = \int_\Omega f v \, \mathrm{d}x. \tag{2}$$

See, for example, Ciarlet [2002], for further details.

On a mesh cell T, the element matrix A_T is computed by evaluating integrals over T (for each pair of i, j),

$$(A_T)_{i,j} = \int_T \kappa \nabla \psi_i^T \cdot \nabla \phi_j^T \, \mathrm{d}x, \tag{3}$$

where ψ_i^T and ϕ_j^T are non-zero local shape functions on T (also called local test and trial functions). Each value in the element vector b_T is calculated by $\int_T f \psi_i^T \, \mathrm{d}x$.

To program the finite element assembly, a computational kernel must be coded to compute the element matrix and vector for each mesh cell. Such a kernel can either be hand-written for each variational form, or, be automatically generated from a high-level description.

2.2 Unified Form Language

The *Unified Form Language (UFL)* [Alnæs et al. 2014] is a high-level language used by both the PDE solver frameworks FEniCS and Firedrake for specifying variational forms in a manner close to their mathematical description. UFL allows defining variational forms in terms of integrals over cells, interior faces, or boundary faces of a mesh. The user indicates whether an integral is to be taken over the whole mesh or some user-defined region. Furthermore, function spaces based on various finite elements [Arnold and Logg 2014] may be prescribed for test and trial spaces, as well as for any coefficients that may be involved.

UFL is a domain-specific language embedded in Python, so users can take advantage of Python's features when writing variational forms. This becomes particularly useful in more advanced PDE solver applications. The code in Algorithm 1 shows UFL descriptions of the variational form for the Poisson problem. Here, linear Lagrange elements on tetrahedral mesh cells are prescribed for the test and trial functions `u` and `v` and coefficient `f`. After defining these functions, the bilinear and linear forms in Eq. (2) are spelled out using **`grad`** and **`inner`** to indicate the gradient operator and dot product.

2.3 Compiling variational forms

The *FEniCS form compiler*, or *FFC*, is responsible for translating a variational form written in UFL into kernels that compute element vectors and matrices. FFC follows standard principles of compiler design, with stages for parsing and analysis, followed by optimisation and code generation based on suitable intermediate representations. Further details are found in the FEniCS book [Logg et al. 2012] or in references given in Section 6.

```
element = FiniteElement("Lagrange",tetrahedron,1)
coords = VectorElement("Lagrange",tetrahedron,1)
mesh = Mesh(coords)
V = FunctionSpace(mesh, element)
u = TrialFunction(V)
v = TestFunction(V)
f = Coefficient(V)
kappa = Constant(mesh)

a = kappa * inner(grad(u), grad(v)) * dx
L = inner(f, v) * dx
```

Algorithm 1. Variational form for Poisson's equation expressed in Unified Form Language.

Most users interact with FEniCS through its Python API. When operating in this mode, UFL forms are embedded in a Python program. Internally, FEniCS automatically generates C code for finite element assembly kernels from those UFL forms, before invoking a C compiler to compile the generated kernels into a shared library. The shared library is then loaded and the compiled kernels can thereafter be executed to assemble linear systems.

Alternatively, FEniCS can be used as a library from a C++ application, still providing the same functionality that is available through Python. However, in this case, UFL forms are placed in standalone UFL scripts (in Python syntax). The objective of these scripts is to define one or more UFL forms. The user is then responsible for invoking the form compiler on these scripts before compiling and linking the generated code to their application.

In this paper, we use FEniCS in the second manner described above, mostly because it is more amenable to carrying out performance benchmarks, such as those presented in Section 5.

2.4 GPU computing

Compared to traditional CPUs, GPUs use a larger portion of their transistors on compute units and registers, and fewer transistors on cache, branch prediction and out-of-order execution. Memory latencies are primarily hidden by using a large number of concurrent threads rather than cache. Consequently, GPUs offer a much higher parallel performance but a far lower sequential one. Scientific computations, such as finite element assembly, often involve applying the same computation to a large number of mesh elements thereby allowing for very wide parallelism and thus GPU acceleration.

While there are several ways of programming GPUs, CUDA [NVIDIA Corporation 2019a] is by far the most common as of now. CUDA extends C++ and FORTRAN to allow offloading subprograms, called kernels, from a host CPU by launching them on a device (i.e., GPU). Typically, a kernel launch contains a large number of concurrent threads, each corresponding roughly to a loop iteration on a CPU. Parallel execution of these threads is scheduled by the GPU hardware.

1. Partition mesh $\mathcal{T}$ into $\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_{P-1}$
2. **for** $p = 0, 1, \ldots, P-1$ **in parallel do**
3. **for all** mesh cells $T \in \mathcal{T}_p$ **do**
4. $x \leftarrow$ `cell_coords`(T) — *Vertex coordinates*
5. $c \leftarrow$ `cell_coeffs`(T) — *Coefficients*
6. $\mu, \nu \leftarrow$ `cell_mapping`(T) — *Local-to-global mappings*
7. $A_e \leftarrow$ `tabulate_tensor`(x, c) — *Compute element matrix*
8. **for** $j = 0$ **to** $m-1$ **and** $k = 0$ **to** $n-1$ **do**
9. **if** $\mu(j)$ or $\nu(k)$ subject to boundary conditions **then**
10. $(A_e)_{j,k} \leftarrow 0$
11. **end if**
12. **end for**
13. `MatSetValues`(A, A_e, μ, ν) — *Add A_e to global matrix A*
14. **end for**
15. **end for**

Algorithm 2. Pseudocode of a CPU-based algorithm for finite element assembly of a global matrix A from a bilinear form.

Though host and device memory are separate, CUDA allows accessing both through a shared address space. Thus, a pointer implicitly carries information on whether it points to host or device memory. CUDA also offers routines for explicitly moving data between host and device memory. If a particular host-side array is repeatedly involved in data transfers, then it is recommended to use *page-locked* (or *pinned*) memory. Doing so ensures that no page faults are caused by copying to or from the page-locked host memory. As a consequence, the CUDA runtime copies data directly to its destination without using an intermediate buffer, significantly improving the effective bandwidth of data transfers between host and device.

As CUDA is proprietary, it is restricted to NVIDIA GPUs. With the increased popularity of GPUs by other vendors, platform-independent alternatives such as OpenCL become more and more interesting.

3 GPU implementation of finite element assembly

In this section, we begin with the existing CPU-based assembly algorithm in FEniCS, and then explain how the form compiler is extended to generate CUDA C++ kernels, based on two algorithmic approaches, for GPU-based finite element assembly.

3.1 Baseline CPU implementation

As the starting point, let us briefly discuss the current CPU-based finite element assembly algorithm used in FEniCS. Algorithm 2 shows a high-level pseudocode for the case of assembling a global coefficient matrix from a bilinear form.

Note that prior to assembly, FEniCS partitions the computational mesh, assigning each part to a different processor. The global vector or matrix can thus be distributed

among the participating processors. Each processor then performs assembly over its portion of the mesh and communicates with others using MPI (when needed).

If a linear or bilinear form consists of more than one integral, then assembly is performed for one integral at a time. For each integral, a loop is carried out over a set of mesh entities, most often the cells of the mesh. It is sometimes necessary to loop over boundary faces to handle boundary conditions.

For each mesh cell, the first step is to read from memory the vertex coordinates of the mesh cell. It may also be necessary to fetch some problem-dependent constants or coefficient values. Second, an element vector or matrix is computed by invoking the *tabulate tensor* kernel that was generated by the form compiler. This kernel uses the cell's vertex coordinates and coefficients, plus the PDE-specific details.

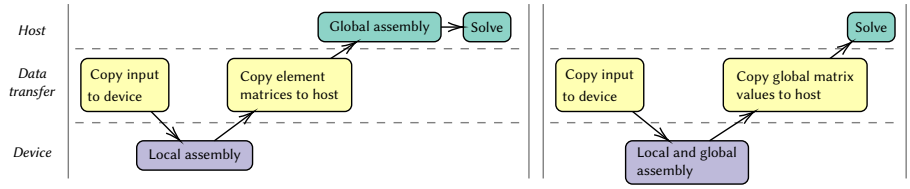
Once an element vector or matrix has been computed, its values are added to the global vector or matrix. In the case of a vector, each entry of an element vector corresponds to an entry in the global vector as determined by a local-to-global mapping. Otherwise, for a bilinear form, each entry of an element matrix corresponds to a row and a column of the global matrix as determined by local-to-global mappings of the test and trial spaces. So an additional step is needed to locate the correct position in the array of non-zero matrix values of the global, sparse matrix. The current assembly algorithm of FEniCS delegates this step to *PETSc* [Balay et al. 2019], a library that FEniCS generally uses to handle sparse matrices and linear solvers. Specifically, element matrix values are added to the global matrix by calling PETSc's `MatSetValues` function, which finds the location of each non-zero value by performing a binary search among the column indices of the relevant row in the global matrix.

There are some additional details concerning Dirichlet boundary conditions. First, element vector or matrix values subject to Dirichlet conditions are set to zero before being added to the global vector or matrix. Second, after the assembly of all the form integrals, the global matrix is modified by setting the diagonal equal to one for rows where Dirichlet conditions apply.

3.2 CUDA kernels for element vector and matrix

Recall that the form compiler, FFC, is responsible for translating a variational form written in UFL into a *tabulate tensor* kernel needed for computing element vectors or matrices. Our first step towards offloading assembly to a CUDA device is to extend FFC to emit a *tabulate tensor* kernel compatible with CUDA C++. Since the PDE-specific variational forms are usually provided at runtime, we use NVIDIA's runtime compilation API (NVRTC) [NVIDIA Corporation 2019b]. Given a string of CUDA C++ source code, NVRTC produces compiled and optimised assembly language code for the PTX instruction set architecture. The PTX assembly is then loaded using the CUDA driver API [NVIDIA Corporation 2019a] by calling the function `cuModuleLoadDataEx`, which further compiles PTX assembly to CUDA device code for a specific GPU model. Once a module is loaded, one can obtain device-side functions annotated with the `__global__` keyword, and the host may launch CUDA kernels to execute those functions on a CUDA device.

Luckily, the C code generated by the original FFC for the usual *tabulate tensor* kernel is almost compatible with CUDA C++. Only minor adaptations are needed



(a) *GPU-based local assembly* **(b)** *GPU-based global assembly*

Figure 1. Diagram of host- and device-side operations and memory transfers for two GPU-based assembly algorithms.

to satisfy the NVRTC compiler (otherwise some standard library headers cannot be directly included in the source code passed to NVRTC). Moreover, the C code is already sufficiently optimised to be used directly for GPU-based assembly.

3.3 GPU-based local assembly

The simplest strategy for migrating assembly to a GPU is to offload only the computation of element matrices/vectors, keeping PETSc to add the computed values to a global matrix/vector on the host. Because each element vector or matrix is independent, this "local" assembly is easily parallelised. The left side of Figure 1 illustrates how the work is divided between the host and device.

Before assembly, the required input data is copied from host memory to device memory, including vertex coordinates of the mesh, local-to-global mappings, boundary markers for Dirichlet boundary conditions, etc. The majority of this data usually does not change over the course of an application that repeatedly assembles linear systems. Thus, a large part of the cost associated with these transfers is paid only once.

As shown in Algorithm 3, the auto-generated main assembly kernel is a *grid-stride loop*, where thread i computes element vectors or matrices for cells i, $i + N$, $i + 2N$, and so on, with N being the total number of threads. The cell vertex coordinates are stored in per-thread arrays with automatic storage duration, which means that the CUDA compiler will try to place the data in registers. If that is not possible, local memory is used, potentially transferring to and from GPU device memory. Each call to `tabulate_tensor` computes an element matrix, placing the computed values in global memory. Moreover, element matrix values are set to zero whenever Dirichlet boundary conditions apply. Afterwards, element matrices are transferred from device memory to host memory, where each element matrix is added to the global matrix by calling the PETSc function `MatSetValues`.

Discussion

The main drawback of offloading local assembly to a GPU and performing global assembly on the host is the intermediate transfer of element matrices/vectors from device to host, moving a large volume of data over a connection with relatively low bandwidth. For example, a mesh with ten million cells and a discretisation with 4×4

```
void __global__ cuda_local_assembly(
  int num_active_cells,
  const int * active_cells,
  const int * vertices_per_cell,
  const double * vertex_coords,
  int num_coeffs_per_cell,
  const double * coeffs, const double * constants,
  const int * dofmap0, const int * dofmap1,
  const char * bc0, const char * bc1,
  double * values)
{
  for (int i = blockIdx.x*blockDim.x+threadIdx.x;
    i < num_active_cells;
    i += blockDim.x * gridDim.x)
  {
    // Set element matrix values to zero
    double* Ae = &values[i*4*4];
    for (int j = 0; j < 4; j++) {
      for (int k = 0; k < 4; k++)
        Ae[j*4+k] = 0.0;
    }

    // Gather cell vertex coords and coefficients
    int c = active_cells[i];
    double cell_vertex_coords[4*3];
    for (int j = 0; j < 4; j++) {
      int vertex = vertices_per_cell[c*4+j];
      for (int k = 0; k < 3; k++) {
        cell_vertex_coords[j*3+k] =
          vertex_coords[vertex*3+k];
      }
    }
    const double * cell_coeffs = &coeffs[
      c*num_coeffs_per_cell];

    // Compute element matrix
    tabulate_tensor(
      Ae, cell_coeffs, constants,
      cell_vertex_coords);

    // Handle Dirichlet boundary conditions
    (...)
  }
}
```

Algorithm 3. Generated CUDA C++ code for the local assembly approach. This example uses 4×4 element matrices, corresponding to first-order elements on a tetrahedral mesh.

element matrices will lead to transferring 1.28 GB, taking at least 80 ms over a typical 16 GB/s PCIe connection. This is far longer than it takes for the GPU to compute the element matrices themselves.

Moreover, many applications use low-order finite elements for which calculating the element matrices is not very expensive. Instead, most of the time is spent adding values to the global matrix. So it does not pay off to only offload local assembly to GPU. In the next section, we extend FFC to also offload the global assembly, letting the GPU add element matrices/vectors to the global matrix/vector.

On a related note, GPU-based local assembly has previously been used in *matrix-free* methods [Ljungkvist 2014], where global assembly is avoided entirely. This however requires a matrix-free linear solver that runs on the GPU. Matrix-free methods are beyond the scope of this paper, and we focus only on global assembly in the following.

3.4 GPU-based global assembly

To improve the GPU-based local assembly described above, we have further extended the form compiler by a "global" approach. This has the goal of migrating the global assembly also to GPU, as shown on the right-hand side of Figure 1.

If we assume that the global matrix is stored in compressed sparse row (CSR) format, then the row pointers and column indices of non-zero matrix entries are copied to device memory prior to assembly. This is done once, since the matrix structure usually does not change. Afterwards, the newly computed global matrix values are copied from device memory to host memory. The amount of data transferred depends on the connectivity of the mesh and the particular discretisation used, but it is always much smaller than the local assembly approach. In Section 4.3, we discuss scenarios where even copying the global matrix values can be avoided.

The auto-generated global assembly kernel is shown in Algorithm 4, with two main differences compared to Algorithm 3. First, element matrices are now stored in per-thread arrays with automatic storage duration, meaning that the compiler will use registers or local memory. Second, for each cell, a final step is carried out, where a binary search is used for each entry of the element matrix to find the corresponding position in the array of global matrix values. Once the location is found, the element matrix value is added to the global matrix value using an atomic operation, which is necessary to avoid race conditions between threads that may attempt to update the same value simultaneously.

4 Optimisations

In this section, we explore several optimisations for the GPU-based finite element assembly kernels described in the previous section. Our choice of optimisations is primarily guided by profiling assembly kernels for the Poisson problem. We expect most of the ideas discussed here to generalise to assembly kernels for other problems. However, some caution is warranted, since other assembly kernels are likely to have a different balance between computations and memory accesses, rendering some optimisations less helpful.

```
void __global__ cuda_global_assembly(
  (...) // Input arguments (see Algorithm 3)
  const int * row_ptr,
  const int * column_indices,
  double * values)
{
  for (int i = blockIdx.x*blockDim.x+threadIdx.x;
    i < num_active_cells;
    i += blockDim.x * gridDim.x)
  {
    (...) // Set element matrix values to zero
    (...) // Gather vertex coords and coefficients
    (...) // Compute element matrix

    // Add values to global matrix, skipping
    // degrees of freedom subject to Dirichlet cond.
    for (int j = 0; j < 4; j++) {
      int row = dofmap0[c*4+j];
      if (bc0 && bc0[row]) continue;
      for (int k = 0; k < 4; k++) {
        int column = dofmap1[c*4+k];
        if (bc1 && bc1[column]) continue;
        int r = binary_search(
          row_ptr[row+1] - row_ptr[row],
          &column_indices[row_ptr[row]], column);
        r += row_ptr[row];
        atomicAdd(&values[r], Ae[j*4+k]);
      }
    }
  }
}
```

Algorithm 4. Generated CUDA C++ code for global matrix assembly. Some parts of the kernel are identical to Algorithm 3 and have therefore been abbreviated.

4.1 Lookup table for global matrix values

Investigating the GPU-based global assembly kernel for the Poisson problem, the NVIDIA profiler clearly shows that performance is limited by memory latency, rather than computations or memory bandwidth. In fact, the profiler reveals that for 95 % of program counter samples retrieved during execution, threads are stalled waiting on memory dependencies. For this kernel, the observed latency is closely correlated with thread divergence caused by branching during binary searches and handling boundary conditions.

In most applications, assembly is carried out repeatedly with local-to-global mappings and boundary condition markers remaining the same between iterations of a non-linear solver or time steps in a time-dependent problem. In this case, it suffices to perform binary searches for element matrix entries only once as a precomputation, storing the results in a lookup table. During assembly, the lookup table (`nonzero_locations` in Algorithm 5) is consulted to find the locations of global matrix non-zeros. Both the computation of the lookup table and the global assembly routine are automatically generated by our extended form compiler.

The lookup table itself requires an additional $4Mn^2$ bytes to be read from global memory, where M is the number of mesh cells and n is the number of rows and columns of an element matrix. On the other hand, it is no longer necessary to read local-to-global mappings, markers for Dirichlet boundary conditions, row pointers or column indices of the global matrix, which amount to $4N$, N, $4(N+1)$, and $4K$ bytes, respectively, where N is the number of rows and columns and K is the number of non-zeros of the global matrix. Moreover, accesses to the lookup table are easily coalesced and therefore enjoy the high bandwidth of the device memory.

4.2 Rowwise assembly

Based on an idea explored by Cecka et al. [2010], we also automatically generate code for a variant of the GPU-based global assembly algorithm, where assembly is performed row by row with respect to the global matrix, rather than cellwise. The resulting algorithm, shown in Algorithm 5, requires a mapping from the global degrees of freedom of the test space to the mesh cells that contain them. A drawback of rowwise assembly is that some redundant operations are carried out whenever a thread computes an element matrix for a given row and cell, but discards computed values unrelated to the current row. However, the advantage is that accesses to the global matrix values are more regular compared to cellwise assembly, and therefore usually faster overall.

4.3 Avoiding CPU-GPU communication

GPU computing inevitably requires copying data between host and device. Since these transfers are time consuming, it is crucial to avoid such data transfers wherever possible to prevent them from becoming a performance bottleneck. For example, if a linear system is assembled on a GPU and the subsequent linear solver also runs on

```
void __global__ cuda_rowwise_assembly(
  (...) // Input arguments (see Algorithm 3)
  const int * cells_per_dof_ptr,
  const int * cells_per_dof,
  const int * nonzero_locations,   // Lookup table
  const int * element_matrix_rows,
  int num_rows,
  double * values)
{
  for (int p = blockIdx.x*blockDim.x+threadIdx.x;
       p < cells_per_dof_ptr[num_rows];
       p += blockDim.x * gridDim.x)
  {
    (...) // Set element matrix values to zero
    (...) // Gather vertex coords and coefficients
    (...) // Compute element matrix

    // Add values to the global matrix, skipping
    // degrees of freedom subject to Dirichlet cond.
    for (int j = 0; j < 4; j++) {
      if (j != element_matrix_rows[p]) continue;
      for (int k = 0; k < 4; k++) {
        int l = ((p/warpSize)*4+k) * warpSize +
          p % warpSize;
        int r = nonzero_locations[l];
        if (r < 0) continue;
        atomicAdd(&values[r], Ae[j*4+k]);
      }
    }
  }
}
```

Algorithm 5. Generated CUDA C++ code for rowwise assembly of a global matrix.

the same device, then there is no need to copy the matrix or vector to the host in the meantime.

Unfortunately, PETSc, which FEniCS relies on for its linear solvers, does not entirely support this mode of operation at the moment. Although PETSc provides an API for accessing pointers to CUDA device memory for its CUDA-based vectors, there is no such API available for accessing pointers to the non-zero values of sparse matrices stored on the device. To be more specific, suppose one of PETSc's CUDA-based linear solvers is to be used and that an array of non-zero matrix values is computed on a CUDA device, for instance, using one of the GPU-based assembly algorithms presented here. To provide PETSc with the newly computed matrix values, it is necessary to first copy them from GPU device memory to host memory before PETSc once more copies the same values from host memory back to the device to update its internal representation.

Table 1. Hardware used in our experiments.

	NVIDIA V100	Intel Xeon Gold 6130	AMD Epyc 7601
Instruction set	PTX	x86-64	x86-64
Microarchitecture	Volta	Skylake (server)	Zen
SMs/CPU cores	80	32	64
Core frequency	1.46 GHz	1.9 to 3.6 GHz	2.7 to 3.2 GHz
DP FLOP rate	7 450 Gflop/s	1 946 Gflop/s	1 382 Gflop/s
Max. bandwidth	898 GB/s	256 GB/s	342 GB/s
STREAM Triad bandwidth	887 GB/s	147.1 GB/s	161.4 GB/s

The cost of copying data in this manner is illustrated in numerical experiments in Section 5.2, where we show that it, in fact, dominates the assembly performance. To avoid this issue, we added functions to PETSc that allow the user to directly access pointers to device-side storage of matrix values for PETSc's CUDA-based sparse matrices.

5 Numerical experiments

In this section, we measure our GPU-based finite element assembly implementations to highlight strengths and weaknesses of various approaches and the effectiveness of the proposed optimisations. We also validate our GPU-enabled form compiler using a non-linear solid mechanics problem and a detailed comparison of CPU and GPU performance.

5.1 Experimental setup

Hardware

Our GPU experiments were carried out on an NVIDIA V100 GPU that belongs to an NVIDIA DGX-2 system. The DGX-2 consists of two Intel Xeon Platinum 8168 CPUs and sixteen V100 GPUs. In addition, some experiments were carried out on two multi-core CPU systems: a dual-socket Intel Xeon Gold 6130 and a dual-socket AMD Epyc 7601. An overview is found in Table 1, including measurements of achievable bandwidth based on BabelStream [Deakin et al. 2018] for NVIDIA V100 and the STREAM benchmark [McCalpin 2013] for Intel Xeon and AMD Epyc. Throughout the benchmarks presented here, we use CUDA 10.1, GCC 9.2.0 and the compiler flags `-O3 -march=native`.

Computational meshes

The following experiments use unstructured, tetrahedral meshes made up of about 6 to 17 million tetrahedral cells, as shown in Table 2. First, there is a standard, uniform mesh of the unit cube, consisting of $100 \times 100 \times 100$ cubes, each subdivided into six

Table 2. Computational meshes used in numerical experiments.

Mesh	Vertices	Cells
Uniform mesh	1 030 301	6 000 000
Cardiac mesh 1	1 255 775	6 735 654
Cardiac mesh 20	3 019 809	16 907 270
Cardiac mesh 41	2 226 802	12 255 517
Cardiac mesh 44	1 958 816	10 697 116

Table 3. Performance (in Mcell/s) of assembling a matrix for the Poisson problem.

Mesh	Xeon	Epyc	Local	Global	Lookup table	Rowwise
Uniform mesh	41.1	58.4	3.7	1100.5	1336.9	1626.9
Cardiac mesh 1	35.4	56.1	2.1	1232.5	1183.1	1720.0
Cardiac mesh 20	34.6	58.0	2.1	582.5	996.6	1601.4
Cardiac mesh 41	35.1	58.7	1.9	469.7	909.8	1607.5
Cardiac mesh 44	35.0	56.1	2.0	533.7	983.6	1692.3

tetrahedra. Second, we employ meshes from a cardiac modelling application [Marciniak et al. 2017], based on patient data from a Danish study on cardiac disease [Jabbari et al. 2015]. These meshes are more representative of real-world applications involving unstructured meshes.

5.2 GPU-based finite element assembly

In this section, we consider the performance of different GPU-based assembly algorithms for the Poisson problem. The main results are shown in Table 3, which compares the performance of assembling a matrix for the Poisson problem using different CUDA-based assembly algorithms on NVIDIA V100. Performance is reported as the number of mesh cells processed per second, in millions, or Mcell/s. For comparison, the performance of FEniCS's parallel, CPU-based assembly on Xeon and Epyc is also shown.

The ineffectiveness of the local assembly approach is apparent. Its poor performance is attributed to expensive data transfers and that adding element matrices to the global matrix is performed on a single core of the host. In contrast, a tremendous acceleration is achieved by offloading global assembly. With global assembly performance varying from about 450 to 1 250 Mcell/s, the structure of the computational mesh has a significant performance impact. This is at least partly explained by the fact that larger, unstructured meshes, such as Cardiac mesh 20, 41 and 44, are less likely to benefit from caching on NVIDIA V100 when accessing vertex coordinates and global matrix data.

Further, using a lookup table for the sparse matrix non-zero locations leads to a notable speedup of about 1.7 to 1.9 for the larger meshes. Also, a further speedup of

up to 70 % is gained by employing the rowwise assembly algorithm for the Poisson problem.

If global assembly for large meshes does indeed suffer from poor cache reuse and latency induced by stalled threads waiting on memory dependencies, then replacing binary searches and accompanying irregular memory accesses with a lookup table is likely to improve the situation simply because of fewer memory dependencies, thus allowing threads to be scheduled in a way that more latency is covered. For the same reason, rowwise assembly also improves performance due to better cache reuse for the global matrix values.

Finally, for a more fair comparison of GPU and CPU performance, we implemented a simple benchmark, independent of FEniCS, of an OpenMP-parallel, CPU-based assembler for the Poisson problem with first-order elements. This benchmark also uses both the lookup table and rowwise assembly optimisations, which yield significantly improved performance. Using 64 cores on AMD Epyc, the resulting performance for assembling a matrix for *Cardiac mesh 20* is about 390 Mcell/s. Nonetheless, GPU-based assembly on NVIDIA V100 is about four times faster.

Impact of CPU-GPU data transfers

Figure 2 compares the time spent performing assembly with the time required to subsequently transfer data between host and device, as explained in Section 4.3.

Measurements obtained with the NVIDIA profiler indicate that the effective bandwidths of these transfers are 4.6 and 13.0 GB/s for pageable and page-locked memory, respectively. Thus, even in the case of page-locked memory, where the effective bandwidth is quite close to the theoretical maximum bandwidth of 16 GB/s, data transfers nearly triple the overall global assembly time. Even worse, for the optimised, rowwise assembly, transferring global matrix values back and forth would increase the assembly time by a factor of about five.

5.3 Validating automated code generation for GPU-based PDE solvers

Finally, we use the extended form compiler for GPU-based assembly and solution of a real-world, solid mechanics problem described by Ølgaard and Wells [2012]. The physical problem is posed as finding a 3D displacement field for a solid body by minimising the total potential energy for a hyperelastic material model. Due to a limited amount of space, we cannot dive into the mathematical details, but rather refer the reader to Ølgaard and Wells [2012]. However, we point out that the problem features a non-linear, vector PDE with variable coefficients, a mixture of boundary conditions, and an unstructured computational mesh. This illustrates the generality of the implemented form compiler and the automated code generation approach for GPU-based assembly and solution of finite element problems.

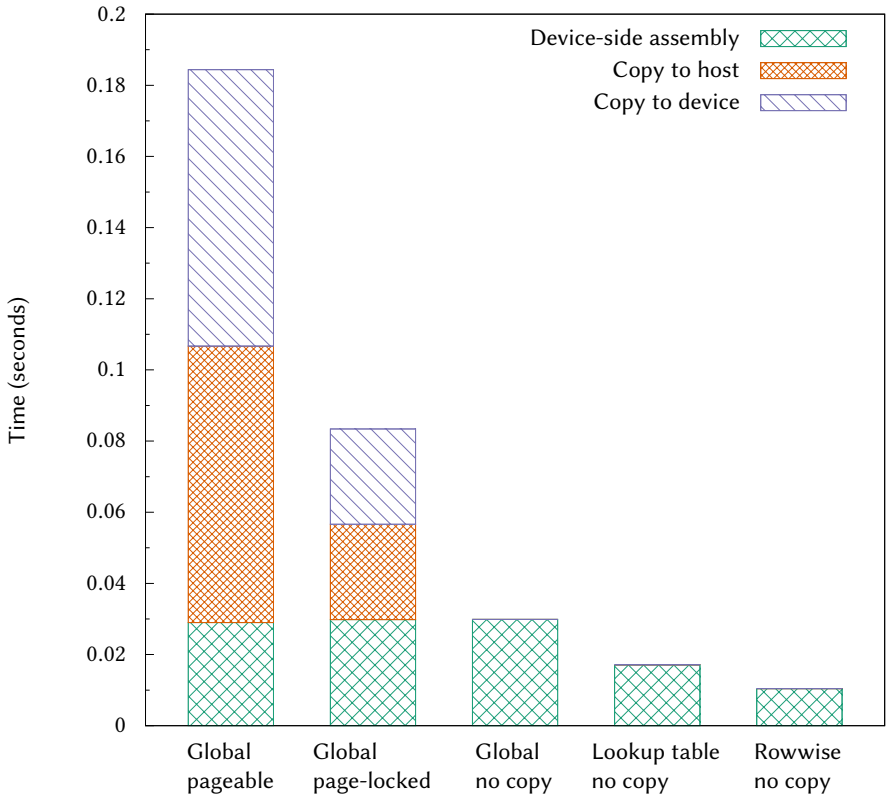


Figure 2. Time (in seconds) spent in assembly and subsequently copying data between GPU device memory and main memory for the Poisson problem with "Cardiac mesh 20" on NVIDIA V100.

Solver

A Newton solver is used to solve the non-linear hyperelasticity problem. The solver converges when the L2-norm of the difference in the approximate solutions from one iteration to the previous iteration falls below a tolerance of 10^{-12}. The linear system that arises during each Newton iteration is solved using the conjugate gradient method with a convergence tolerance of 0.1 for the relative residual, which was found to give the overall fastest solution time. The conjugate gradient solver is implemented by PETSc, which uses NVIDIA's cuSPARSE library [NVIDIA Corporation 2020a] for GPU-accelerated sparse matrix operations. Since rowwise assembly is not currently optimised for the case of a vector PDE, the GPU implementation uses the global assembly algorithm with a lookup table.

Table 4. Performance of assembly and solution of linear systems for a hyperelasticity problem on Intel Xeon Gold 6130, AMD Epyc 7601, and NVIDIA V100.

	Assembly, Mcell/s			Solve, iteration/s		
Mesh	Xeon	Epyc	V100	Xeon	Epyc	V100
Uniform mesh	3.50	5.24	35.25	78.57	157.91	328.85
Cardiac mesh 1	3.88	6.17	31.46	64.34	136.79	279.06
Cardiac mesh 20	3.94	5.71	31.04	25.41	48.19	114.45
Cardiac mesh 41	3.95	4.95	30.73	35.14	69.16	156.71
Cardiac mesh 44	3.96	5.87	30.94	39.58	81.94	178.92

Comparing CPU and GPU performance

Table 4 shows the performance on Intel Xeon Gold 6130, AMD Epyc 7601 and NVIDIA V100 of both assembly and solution stages. The former is reported as the number of mesh cells processed per second, in millions, and the latter is given in iterations per second of the linear solver.

To summarise, NVIDIA V100 demonstrates a speedup of about 8 and 5 over Intel Xeon and AMD Epyc, respectively, during assembly. The speedups for the linear solver range from 4.2 to 4.5 and 2.0 to 2.4 for Intel Xeon and AMD Epyc, respectively.

Figure 3 shows the execution time of the hyperelasticity solver on our three hardware platforms, though we omit initialisation steps that are required regardless of the method, such as loading the mesh and creating sparse matrices. We have however included costs that are incurred specifically due to using the GPU-based solver, such as copying data to GPU device memory and computing the lookup table described in Section 4.1.

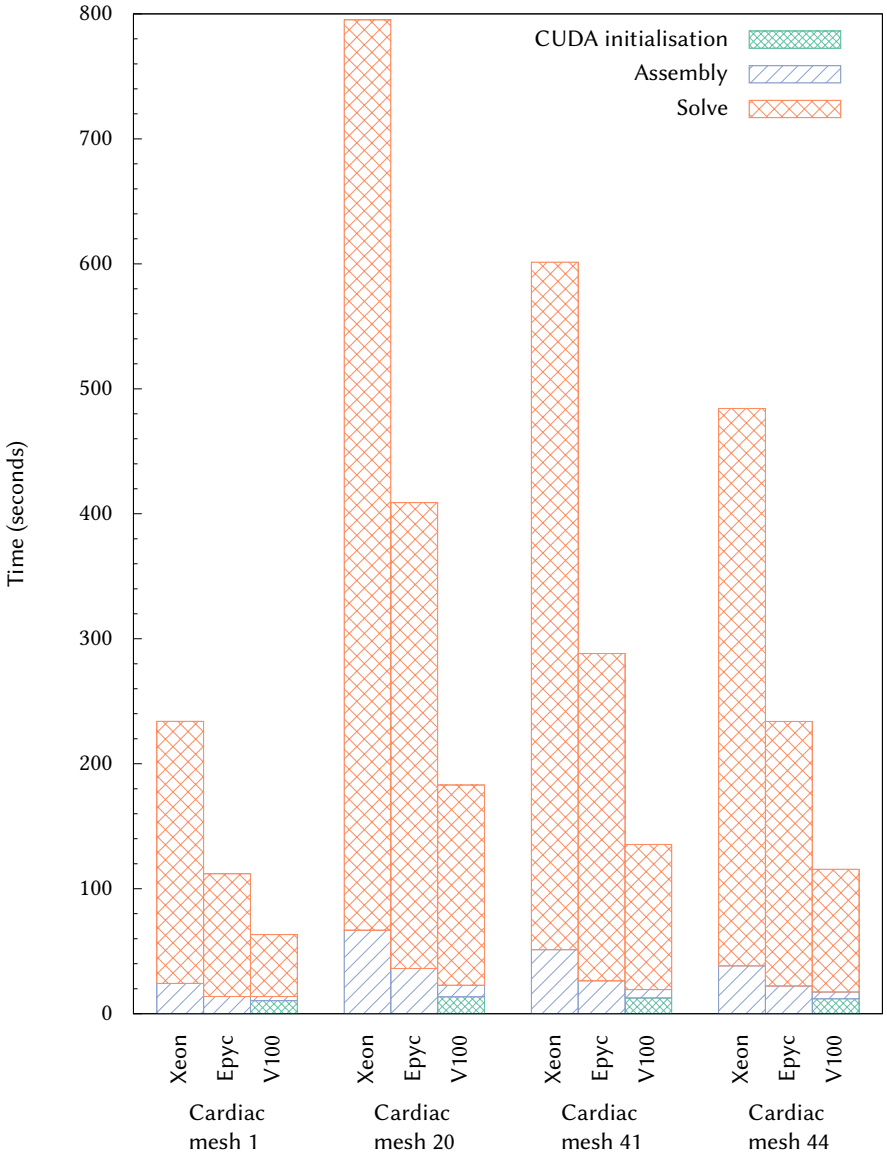
Most of the time is spent by the linear solver, which sees a substantial acceleration from moving to a GPU. This is accredited to the acceleration of matrix-vector multiplications, which tend to be limited by memory bandwidth. Recall that the memory bandwidth of GPU device memory on NVIDIA V100 is considerably higher than the memory bandwidth on either of the multi-core CPU platforms.

Assembling linear systems on NVIDIA V100 is also noticeably faster than on the CPUs. Although the initial costs related to performing assembly on the GPU are fairly high in this example, these are likely to become negligible in the case of a time-dependent problem.

6 Related work

The implementation of FEniCS is described in the FEniCS book [Logg et al. 2012], alongside several applications to solving PDEs. During the course of the development of FFC and other, related form compilers, numerous strategies have been explored with regards to generating and optimising code for computing element vectors and matrices. For example, a tensor representation [Kirby et al. 2005; Kirby and Logg 2006; Rognes et al. 2009] or computer algebra [Alnæs and Mardal 2010; Russell and Kelly 2013] can

Figure 3. Execution time (in seconds) for solving a hyperelasticity problem on Intel Xeon Gold 6130, AMD Epyc 7601, and NVIDIA V100.



sometimes be used to simplify or compute integrals exactly. Other approaches include optimising kernels based on numerical integration [Ølgaard and Wells 2010], various low-level loop optimisations [Homolya et al. 2018; Luporini et al. 2017; Luporini et al. 2015], or explicit vectorisation [Sun et al. 2020]. In addition, Knepley and Terrel [2013] and Banaś et al. [2014] were able to optimise GPU kernels for computing element matrices through well-considered assignment of work to different threads and memory layout for data in global memory, as well as the use of shared memory to speed up certain memory accesses.

Cecka et al. [2010] studied several different GPU-based algorithms for global assembly for the Poisson problem, including the usual cellwise algorithm, a rowwise algorithm, which we adopted in Section 4.2, and a third algorithm that assigns each non-zero of the global matrix to a separate thread. Experiments show that the third method can be worthwhile, though there are challenges related to limited availability of shared memory, load imbalance, and a large number of redundant operations being carried out.

For a time-dependent, non-linear diffusion equation, Pichler and Haase [2017] offload assembly to a GPU using the CUDA C++ template library Thrust [NVIDIA Corporation 2020b], whose high-level interface is used to avoid writing CUDA kernels. Moreover, global assembly is carried out using a precomputed lookup table to add element matrix values to a global matrix, in the same way that is described in Section 4.1.

The translation from Unified Form Language to low-level kernels for GPUs has previously been considered by Markall et al. [2010], where a prototype compiler is discussed. At the same time, the authors also advocate bypassing global assembly to avoid the costly process of searching for global matrix non-zeros during assembly (see also Markall et al. [2013]). The result, which is called a *local matrix approach*, is similar to a matrix-free method, although element matrices are explicitly stored in GPU device memory instead of being recomputed as they are needed.

An example of GPU acceleration of both assembly and solution of linear systems for a finite element problem is given by Fu et al. [2014], including a more advanced linear solver based on the conjugate gradient method with an algebraic multigrid preconditioner. Reguly and Giles [2015] thoroughly investigate GPU acceleration for both assembly and solution of linear systems for the Poisson problem, comparing global assembly using different sparse matrix formats to the local matrix approach and matrix-free methods. Roughly speaking, the linear solver benefits from global assembly whenever first-order elements are used, while other strategies may be advantageous for second-, third- and fourth-order elements. Further research on matrix-free methods has been conducted by Ljungkvist [2014] and Kronbichler and Ljungkvist [2019].

7 Conclusion

We have enabled automated generation of GPU code for finite element assembly, which lays the foundation for fully GPU-accelerated PDE solvers in FEniCS. The GPU-accelerated solution of a real-world non-linear PDE demonstrates the achievable

speedup, while still retaining the ease of use offered by UFL as a high-level, domain-specific language for finite element problems.

Furthermore, global assembly can be optimised by replacing costly binary searches with the use of a precomputed lookup table. Performing the assembly row by row is also shown to improve performance for the Poisson problem with first-order elements.

We stress the need for paying careful attention to CPU-GPU data transfers to successfully accelerate finite element computations. Otherwise, the benefits of GPU-based assembly can easily dissipate. Our strategy is different from that chosen by many other libraries, which is based on separately offloading the assembly and solution phases to a GPU. Indeed, a high-level view of the entire PDE solving procedure is necessary to benefit from seamlessly integrated GPU acceleration.

Users of the FEniCS framework can use our GPU-enabled form compiler to access the other advanced features of FEniCS, but also benefit from GPU-accelerated computations.

Acknowledgements

This work was supported by the Research Council of Norway under contract 251186. Also, the research presented in this paper has benefited from the Experimental Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053.

References

Alnæs, M. S., A. Logg, K. B. Ølgaard, M. B. Rognes, and G. N. Wells (Feb. 2014). "Unified form language: A domain-specific language for weak formulations of partial differential equations". In: *ACM Trans. Math. Softw.* 40.2. ISSN: 0098-3500. DOI: 10.1145/2566630.

Alnæs, M. S. and K.-A. Mardal (Jan. 2010). "On the Efficiency of Symbolic Computations Combined with Code Generation for Finite Element Methods". In: *ACM Trans. Math. Softw.* 37.1. ISSN: 0098-3500. DOI: 10.1145/1644001.1644007.

Anzt, H., E. Boman, R. Falgout, P. Ghysels, M. Heroux, X. Li, L. Curfman McInnes, R. Tran Mills, S. Rajamanickam, K. Rupp, B. Smith, I. Yamazaki, and U. Meier Yang (2020). "Preparing sparse solvers for exascale computing". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378.2166. DOI: 10.1098/rsta.2019.0053.

Arnold, D. N. and A. Logg (Nov. 2014). "Periodic Table of the Finite Elements". In: *SIAM News* 47 (9). URL: http://www.femtable.org/.

Balay, S., S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. Gropp, D. Karpeyev, D. Kaushik, M. Knepley, D. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. Smith, S. Zampini, H. Zhang, and H. Zhang (2019). *PETSc Web page*. URL: https://www.mcs.anl.gov/petsc.

Banaś, K., P. Płaszewski, and P. Macioł (2014). "Numerical integration on GPUs for higher order finite elements". In: *Computers & Mathematics with Applications* 67.6, pp. 1319–1344. ISSN: 0898-1221. DOI: 10.1016/j.camwa.2014.01.021.

Cecka, C., A. J. Lew, and E. Darve (Aug. 2010). "Assembly of finite element methods on graphics processors". In: *International Journal for Numerical Methods in Engineering* 85.5, pp. 640–669. DOI: 10.1002/nme.2989.

Ciarlet, P. G. (2002). *The Finite Element Method for Elliptic Problems*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics. ISBN: 0-89871-514-8.

Deakin, T., J. Price, M. Martineau, and S. McIntosh-Smith (2018). "Evaluating attainable memory bandwidth of parallel programming models via BabelStream". In: *International Journal of Computational Science and Engineering* 17.3, pp. 247–262. DOI: 10.1504/IJCSE.2018.095847.

Fu, Z., T. J. Lewis, R. M. Kirby, and R. T. Whitaker (Feb. 2014). "Architecting the finite element method pipeline for the GPU". In: *Journal of Computational and Applied Mathematics* 257, pp. 195–211. DOI: 10.1016/j.cam.2013.09.001.

Homolya, M., L. Mitchell, F. Luporini, and D. A. Ham (June 2018). "TSFC: A Structure-Preserving Form Compiler". In: *SIAM Journal on Scientific Computing* 40.3, pp. 401–428. ISSN: 1064-8275. DOI: 10.1137/17M1130642.

Jabbari, R., T. Engstrøm, C. Glinge, B. Risgaard, J. Jabbari, B. G. Winkel, C. J. Terkelsen, H.-H. Tilsted, L. O. Jensen, M. Hougaard, S. E. Chiuve, F. Pedersen, J. H. Svendsen, S. Haunsø, C. M. Albert, and J. Tfelt-Hansen (Jan. 2015). "Incidence and risk factors of ventricular fibrillation before primary angioplasty in patients with first ST-elevation myocardial infarction: a nationwide study in Denmark". In: *Journal of the American Heart Association* 4.1. ISSN: 2047-9980. DOI: 10.1161/JAHA.114.001399.

Kirby, R. C., M. Knepley, A. Logg, and L. R. Scott (2005). "Optimizing the Evaluation of Finite Element Matrices". In: *SIAM Journal on Scientific Computing* 27.3, pp. 741–758. ISSN: 1064-8275. DOI: 10.1137/040607824.

Kirby, R. C. and A. Logg (Sept. 2006). "A compiler for variational forms". In: *ACM Trans. Math. Softw.* 32.3, pp. 417–444. ISSN: 0098-3500. DOI: 10.1145/1163641.1163644.

Knepley, M. G. and A. R. Terrel (2013). "Finite Element Integration on GPUs". In: *ACM Transactions on Mathematical Software* 39.2. DOI: 10.1145/2427023.2427027.

Kronbichler, M. and K. Ljungkvist (May 2019). "Multigrid for Matrix-Free High-Order Finite Element Computations on Graphics Processors". In: *ACM Trans. Parallel Comput.* 6.1. ISSN: 2329-4949. DOI: 10.1145/3322813.

Ljungkvist, K. (2014). "Matrix-Free Finite-Element Operator Application on Graphics Processing Units". In: *Euro-Par 2014: Parallel Processing Workshops*. Ed. by L. Lopes, J. Žilinskas, A. Costan, R. G. Cascella, G. Kecskemeti, E. Jeannot, M. Cannataro, L. Ricci, S. Benkner, S. Petit, V. Scarano, J. Gracia, S. Hunold, S. L. Scott, S. Lankes, C. Lengauer, J. Carretero, J. Breitbart, and M. Alexander. Springer International Publishing, pp. 450–461. ISBN: 978-3-319-14313-2. DOI: 10.1007/978-3-319-14313-2_38.

Logg, A., K.-A. Mardal, G. N. Wells, et al. (2012). *Automated Solution of Differential Equations by the Finite Element Method*. Berlin: Springer. ISBN: 978-3-642-23098-1. DOI: 10.1007/978-3-642-23099-8.

Luporini, F., D. A. Ham, and P. H. J. Kelly (Mar. 2017). "An Algorithm for the Optimization of Finite Element Integration Loops". In: *ACM Transactions on Mathematical Software* 44.1. ISSN: 0098-3500. DOI: 10.1145/3054944.

Luporini, F., A. L. Varbanescu, F. Rathgeber, G.-T. Bercea, J. Ramanujam, D. A. Ham, and P. H. J. Kelly (Jan. 2015). "Cross-Loop Optimization of Arithmetic Intensity for

Finite Element Local Assembly". In: *ACM Transactions on Architecture and Code Optimization* 11.4. ISSN: 1544-3566. DOI: 10.1145/2687415.

Marciniak, M., H. Arevalo, J. Tfelt-Hansen, T. Jespersen, R. Jabbari, C. Glinge, K. A. Ahtarovski, N. Vejlstrup, T. Engstrom, M. M. Maleckar, and K. McLeod (Jan. 2017). "From CMR Image to Patient-Specific Simulation and Population-Based Analysis: Tutorial for an Openly Available Image-Processing Pipeline". In: *STACOM 2016: Statistical Atlases and Computational Models of the Heart. Imaging and Modelling Challenges*. Ed. by T. Mansi, K. McLeod, M. Pop, K. Rhode, M. Sermesant, and A. Young. Springer International Publishing, pp. 106–117. ISBN: 978-3-319-52718-5. DOI: 10.1007/978-3-319-52718-5_12.

Markall, G. R., A. Slemmer, D. A. Ham, P. H. J. Kelly, C. D. Cantwell, and S. J. Sherwin (Jan. 2013). "Finite element assembly strategies on multi-core and many-core architectures". In: *International Journal for Numerical Methods in Fluids* 71.1, pp. 80–97. ISSN: 0271-2091. DOI: 10.1002/fld.3648.

Markall, G. R., D. A. Ham, and P. H. Kelly (2010). "Towards generating optimised finite element solvers for GPUs from high-level specifications". In: *Procedia Computer Science* 1.1. ICCS 2010, pp. 1815–1823. ISSN: 1877-0509. DOI: 10.1016/j.procs.2010.04.203.

McCalpin, J. D. (Jan. 2013). *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Department of Computer Science School of Engineering and Applied Science, University of Virginia. Charlottesville, Virginia. URL: https://www.cs.virginia.edu/stream/.

NVIDIA Corporation (July 2019a). *CUDA Driver API: API Reference Manual*. NVIDIA Corporation. URL: https://docs.nvidia.com/cuda/cuda-driver-api/.

— (July 2019b). *NVRTC – CUDA runtime compilation user guide*. NVIDIA Corporation. URL: https://docs.nvidia.com/cuda/nvrtc/.

— (July 2020a). *cuSPARSE Library*. NVIDIA Corporation. URL: https://docs.nvidia.com/cuda/cusparse/index.html.

— (Aug. 2020b). *Thrust Quick Start Guide*. NVIDIA Corporation. URL: https://docs.nvidia.com/cuda/thrust/.

Ølgaard, K. B. and G. N. Wells (Jan. 2010). "Optimizations for quadrature representations of finite element tensors through automated code generation". In: *ACM Transactions on Mathematical Software* 37.1. ISSN: 0098-3500. DOI: 10.1145/1644001.1644009.

— (2012). "Applications in solid mechanics". In: *Automated Solution of Differential Equations by the Finite Element Method*. Ed. by A. Logg, K.-A. Mardal, and G. N. Wells. Berlin: Springer. Chap. 26, pp. 505–526. ISBN: 978-3-642-23098-1. DOI: 10.1007/978-3-642-23099-8.

Pichler, F. and G. Haase (Mar. 2017). "Finite element method completely implemented for graphic processor units using parallel algorithm libraries". In: *The International Journal of High Performance Computing Applications* 33.1, pp. 53–66. DOI: 10.1177/1094342017694703.

Rathgeber, F., D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly (Dec. 2016). "Firedrake: Automating the Finite Element Method by Composing Abstractions". In: *ACM Transactions on Mathematical Software* 43.3. ISSN: 0098-3500. DOI: 10.1145/2998441.

Reguly, I. Z. and M. B. Giles (Apr. 2015). "Finite Element Algorithms and Data Structures on Graphical Processing Units". In: *International Journal of Parallel Programming* 43.2, pp. 203–239. DOI: 10.1007/s10766-013-0301-6.

Rognes, M. E., R. C. Kirby, and A. Logg (Nov. 2009). "Efficient Assembly of $H(\mathrm{div})$ and $H(\mathrm{curl})$ Conforming Finite Elements". In: *SIAM Journal on Scientific Computing* 31.6, pp. 4130–4151. ISSN: 1064-8275. DOI: 10.1137/08073901X.

Russell, F. P. and P. H. J. Kelly (July 2013). "Optimized Code Generation for Finite Element Local Assembly Using Symbolic Manipulation". In: *ACM Transactions on Mathematical Software* 39.4. ISSN: 0098-3500. DOI: 10.1145/2491491.2491496.

Sun, T., L. Mitchell, K. Kulkarni, A. Klöckner, D. A. Ham, and P. H. Kelly (July 2020). "A study of vectorization for matrix-free finite element methods". In: *The International Journal of High Performance Computing Applications*. DOI: 10.1177/1094342020945005.