Parallel performance modeling of irregular applications in cell-centered finite volume methods over unstructured tetrahedral meshes

J. Langguth[a,*], N. Wu[a,b], J. Chai[a,b], X. Cai[a,c]

[a]*Simula Research Laboratory, Fornebu, Norway*
[b]*National University of Defense Technology, Changsha, China*
[c]*University of Oslo, Oslo, Norway*

Abstract

Finite volume methods are widely used numerical strategies for solving partial differential equations. This paper aims at obtaining a quantitative understanding of the achievable performance of the cell-centered finite volume method on 3D unstructured tetrahedral meshes, using traditional multicore CPUs as well as modern GPUs. By using an optimized implementation and a synthetic connectivity matrix that exhibits a perfect structure of equal-sized blocks lying on the main diagonal, we can closely relate the achievable computing performance to the size of these diagonal blocks. Moreover, we have derived a theoretical model for identifying characteristic levels of the attainable performance as function of hardware parameters, based on which a realistic upper limit of the performance can be predicted accurately. For real-world tetrahedral meshes, the key to high performance lies in a reordering of the tetrahedra, such that the resulting connectivity matrix resembles a block diagonal form where the optimal size of the blocks depends on the hardware. Numerical experiments confirm that the achieved performance is close to the practically attainable maximum and it reaches 75% of the theoretical upper limit, independent of the actual tetrahedral mesh considered. From this, we develop a general model capable of identifying bottleneck performance of a systems' memory hierarchy in irregular applications.

1. Introduction

For any computer program that implements a numerical computation over an unstructured mesh, irregular accesses to the data structures are unavoidable. Such irregular data accesses put significant pressure on the different levels of the memory hierarchy. The common wisdom for CPU programming is thus

*Corresponding author
Email addresses: `langguth@simula.no` (J. Langguth), `nanwu@nudt.edu.cn` (N. Wu), `chaijun200306@nudt.edu.cn` (J. Chai), `xingca@simula.no` (X. Cai)

to strive for good spatial and temporal locality of data on all cache levels. This strategy becomes even more important on GPUs, because the discrepancy between a GPU's global memory bandwidth and its floating-point capabilities is even wider.

Finite volume methods are widely used numerical strategies for solving partial differential equations. Advantages of using finite volumes include built-in support for conservation laws and applicability for unstructured computational meshes. The cell-centered finite volume method is the most common variant, where the degrees of freedom lie in the center of each computational cell. In this paper, we study the most representative 3D scenario where the computational mesh is irregularly made up of tetrahedra which constitute the computational cells. Our objective is to obtain a *quantitative* understanding of the impact of spatial and temporal data locality on the speed of computations when executed on the Kepler GPU by Nvidia [1] and on a typical dual-socket HPC node equipped with Intel CPUs.

For an unstructured tetrahedral mesh, the only viable data structure for storing the tetrahedron-center values is a 1D array. There is no universally ideal ordering of the tetrahedra. However, a completely random ordering often means poor computing speed, because of the consequent random jumps back and forth in the 1D array. Another important observation for cell-centered finite volume computations over tetrahedral meshes is that each computational cell (except for the boundary cells) is directly coupled with exactly four neighboring cells. This is so because the coupling from one tetrahedron to its neighboring tetrahedra arises from flux-type computations across the four triangular faces. Without loosing generality, it suffices to study the following computation:

$$y(i) = \sum_{j=1}^{4} A(i,j)\left(x(\mathcal{I}(i,j)) - x(i)\right), \tag{1}$$

where x and y denote two 1D arrays that store two sets of tetrahedron-center values. Index i loops from 1 to the total number of tetrahedra n, where $\mathcal{I}(i,\cdot)$ gives the face-to-face connection from tetrahedron i to all its four neighboring tetrahedra. The values of A are weights representing the pairwise tetrahedron-tetrahedron couplings. In terms of data structure, all the entries of A are typically stored in a 2D array, where the first dimension equals the total number of tetrahedra, and the second dimension is four. We also remark that for each tetrahedron, the associated amount of computation is 11 floating-point operations (FLOPS), namely 4 subtractions, 4 multiplications and 3 additions.

In order to perform this computation, 4 entries of A and 5 entries of x must be provided. Throughout this paper, we assume that A is stored in the *ELLpack* matrix format [2]. This means that an additional 4 entries of $\mathcal{I}$ are required. The ELLpack format was shown to work well on GPUs in [3]. Since most tetrahedra have exactly four neighbors, it is more efficient to assume this and use padding if the number of neighbors is lower than to store the number of neighbors explicitly. Furthermore, we assume 64-bit floating point values and 32-bit integers, which implies a computational intensity (i.e. FLOP per byte

ratio) of 0.125, since 88 bytes must be read to perform 11 FLOPS. However, as each entry of x is accessed five times - four times by the four neighboring tetrahedra and once by its tetrahedron owner - proper caching can reduce the number of memory accesses on x from 5 to an average of 1 per tetrahedron. This results in an effective computational intensity relative to memory to 0.196, while the computational intensity relative to the cache traffic remains at 0.125. Because data is moved to cache in entire cache lines, the effective intensity can be even lower. We study this effect in sections 7 and 8.

In addition to reading the data, one y value of 8 bytes must be written back to memory for each tetrahedron, which consumes additional bandwidth on most systems and thus reduces the above intensity to 0.1712 and 0.115 respectively. Depending on the architecture and the size of its cache line, overfetching can occur which in effect further reduces the above values.

In any case, the kernel is severely memory bound and the maximum achievable performance in FLOPS will be far below the theoretical peak performance. In the following, we will investigate the achievable computational intensity relative to the memory traffic and caches on the Nvidia K20 Kepler GPU and on Sandy Bridge Intel CPUs.

The remainder of the paper is organized as follows: in Sections 2 and 3, we describe the hardware and the implementations used for our experiments. Section 4 details the setup of the experiments while Sections 5 and 6 describe the experimental results. Based on these, our performance model for the GPU is detailed in Section 7 and that for the CPU in Section 8. Based on this, we develop a general performance model in Section 9. Finally, we give a brief discussion of related work and our conclusions.

2. GPU Platform and Implementation

The heart of our GPU test implementation consists of two CUDA kernel functions that compute Eq. (1) in a straightforward manner while using the memory hierarchy efficiently. These are detailed in figures 2 and 3 below.

In order to highlight the details, let us first take a look at the architecture and memory hierarchy of the Nvidia GK110 GPU, labeled K20m "Kepler". On each of its 13 streaming multiprocessors, the K20 possesses 48 KB of read-only cache, as well as 64 KB of on-chip storage that is divided between shared memory and level 1 cache (L1). In our experiments 48 KB are assigned to shared memory.

In order to obtain the best performance, it is crucial to optimize the use of this on-chip storage. Accesses to shared memory are fast, but data has to be placed there explicitly. Read-only cache is comparably fast [4] and requires only flagging of variables, but eviction of data from it cannot be controlled by the programmer. In previous generations of Nvidia GPUs, the read-only cache was designated as texture cache and could only be accessed for computation by using unwieldy commands. In CUDA 5.x using devices with compute capability 3.5 however, variables can easily be flagged for read-only caching using the *const __restrict* qualifier, or the *__ldg* instruction. Since *const __restrict* is essentially

only a hint for the compiler to use read-only cache, using *__ldg* is preferable for obtaining more predictable performance.

Data that is not fetched into read-only cache or coalesced using shared memory will be accessed by reading 128 bytes at a time, i.e. the SIMD width of the GPU, which leads to significant overfetching since our application loads at most 88 bytes of data per tetrahedron, and only up to 32 contiguous bytes per array. This overfetching leads to severely reduced performance. On the other hand, L2 and read-only cache is read at 32 bytes at a time, and thus does not suffer from this problem.

We therefore avoid using direct access, which leaves us with two viable memory placement strategies. In the first option, A and $\mathcal{I}$ are placed in the shared memory, while read-only cache is used exclusively for caching the x vector. In the second option, all three arrays are cached in read-only cache. Placing parts of the x vector into shared memory is not viable, since the elements that will be accessed are not known beforehand.

Meanwhile, accesses to A and $\mathcal{I}$ are regular which allows explicitly preloading all required elements into shared memory. When implemented in a straightforward manner, this incurs the same overfetching problems as direct accesses. However, because all threads in a thread block can see the same shared memory contents, it is possible to coalesce these accesses, thereby avoiding overfetching. This is achieved by spreading the memory accesses of an entire thread block in such a way that every thread reads one value at a time, irrespective of which thread will actually use the value. See Kernel 1 in Figure 2 for implementation details.

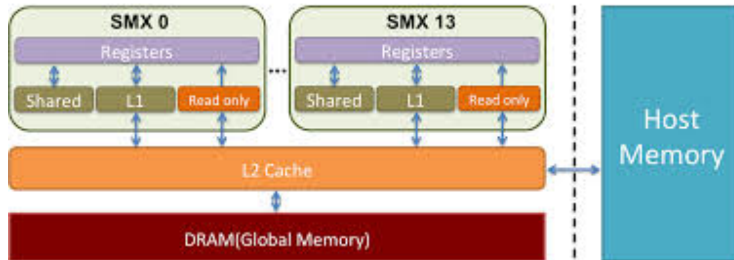

Figure 1: Memory hierarchy of the K20 Kepler GPU. Note that the model used in this paper has only 13 SMX. Source: www.olcf.ornl.gov

Together, L1, read-only cache, and shared memory can be thought of as the first level of cache on the K20 GPU. From the technical point of view, it works in a manner substantially different from a CPU, where L1 cache does not need to be managed explicitly. On the other hand, the second level of cache (L2) is rather similar. The K20m has 1280 KB of L2 cache which is shared among the 13 multiprocessors. All accesses to and from the GPU's global memory are cached via L2 automatically. Thus, its function is similar to the shared third level cache on current multicore CPUs. Accesses to L2 on the K20 always transfer

an entire 32 byte cache line. An overview of the Kepler memory hierarchy is given in Figure 1.

Due to the comparatively large amount of memory traffic, the device needs to launch a large number of parallel threads in order to hide latency. We opt for the simple approach of using one thread per tetrahedron with thread block sizes between 64 and 512. Larger thread block sizes did not result in improved performance, and neither does using kernels that process more than one tetrahedron per thread.

In Figure 2 below, we present Kernel 1, which implements the shared memory based approach. For this Kernel, the number of threads actually running in parallel is bounded by the available shared memory. For example, a thread block of 128 threads requires $128 * 4 * 8 = 4$ KB of shared memory for A and half that amount for $\mathcal{I}$, since the integer values require only 4 bytes each. Thus, only eight such blocks (i.e., 1024 threads) can be run on a streaming multiprocessor in parallel with the available 48 KB of shared memory, even though it is theoretically capable of running 2048 threads at a time.

```
__global__ void Kernel_1
(double* A, int* I, double* X, double* Y)
{
  __shared__ double SM_A[Thread_Block_Size*4];
  __shared__ int    SM_I[Thread_Block_Size*4];
  int b_start=blockIdx.x*blockDim.x*4;
  int tIdx = threadIdx.x;
  int i  = threadIdx.x+blockIdx.x*blockDim.x;
  int l_x = threadIdx.x*4;

  //Load A and I into shared memory
  SM_A[tIdx]             =A[b_start+tIdx];
  SM_A[tIdx+blockDim.x]  =A[b_start+tIdx+blockDim.x];
  SM_A[tIdx+blockDim.x*2]=A[b_start+tIdx+blockDim.x*2];
  SM_A[tIdx+blockDim.x*3]=A[b_start+tIdx+blockDim.x*3];
  SM_I[tIdx]             =I[b_start+tIdx];
  SM_I[tIdx+blockDim.x]  =I[b_start+tIdx+blockDim.x];
  SM_I[tIdx+blockDim.x*2]=I[b_start+tIdx+blockDim.x*2];
  SM_I[tIdx+blockDim.x*3]=I[b_start+tIdx+blockDim.x*3];
  __syncthreads();
  //Main computation
  Y[i] =
         SM_A[l_x+0]*(__ldg(&X[SM_I[l_x+0]])-__ldg(&X[i]))+
         SM_A[l_x+1]*(__ldg(&X[SM_I[l_x+1]])-__ldg(&X[i]))+
         SM_A[l_x+2]*(__ldg(&X[SM_I[l_x+2]])-__ldg(&X[i]))+
         SM_A[l_x+3]*(__ldg(&X[SM_I[l_x+3]])-__ldg(&X[i]));
}
```

Figure 2: CUDA Kernel 1 which uses shared memory to access A and $\mathcal{I}$ in a coalesced manner.

We also implement Kernel 2 shown in Figure 3. It does not have the above

```
__global__ void Kernel_2
(double* A, int* I, double* X, double* Y)
{
  int i  = threadIdx.x+blockIdx.x*blockDim.x;
  int l_x = threadIdx.x*4;
  //Main computation
  Y[i] =
         __ldg(A[l_x+0])*(__ldg(&X[__ldg(I[l_x+0])])-__ldg(&X[i]))+
         __ldg(A[l_x+1])*(__ldg(&X[__ldg(I[l_x+1])])-__ldg(&X[i]))+
         __ldg(A[l_x+2])*(__ldg(&X[__ldg(I[l_x+2])])-__ldg(&X[i]))+
         __ldg(A[l_x+3])*(__ldg(&X[__ldg(I[l_x+3])])-__ldg(&X[i]));
}
```

Figure 3: CUDA Kernel 2 which uses read-only cache to access A and $\mathcal{I}$.

limitation since it exclusively uses read-only cache to access A, $\mathcal{I}$, and x. Kernel 2 is therefore not limited by available shared memory, but due to the limited size of read-only cache, data can be evicted prematurely. See Section 5 for a comparison of both kernels' respective performance. The implementation is straightforward except for the use of the *__ldg* instruction, as shown in Figure 3.

3. Multicore Platform and Implementation

For the multicore test platform, we use dual 8 core Intel E5-2670 processors running at 2.6 GHz. Each core has 64 KB of L1 cache that is divided in 32 KB instruction and 32 KB data cache, and 256 KB of unified L2 cache. Each processor has 20 MB of L3 cache which is shared among its cores. The cacheline size is 64 bytes on all levels of cache. Unlike the GPU, cache is not managed explicitly by the programmer.

Since memory controllers are integrated in the processors, each CPU socket can access only its local memory at full memory bandwidth. The memory of the second socket must be accessed via the QuickPath Interconnect at higher latency, thus resulting in a non-uniform memory access (NUMA) architecture.

We use a simple and a full multicore implementation which are both based on OpenMP. The reason for this is that while the algorithm can be implemented in OpenMP using a simple *parallel for* pragma, doing so prevents it from scaling to two sockets. This is shown in detail in Figure 13 in Section 6. Figure 4 shows the simple implementation. It uses static *parallel for* scheduling in order to parallelize the computation for the tetrahedra. Because presenting full C code would be tedious here, only the salient parts are given in actual code, while the rest is shortened to pseudocode.

In contrast to the simple version, the full implementation explicitly divides the tetrahedra among the threads and copies the required A and $\mathcal{I}$ values into private arrays for each thread. This was necessary to maintain scalability in the NUMA environment. Figure 5 shows the pseudocode. In an initialization step, data is moved to the private arrays. The compute step that follows is the CPU

```
for #timesteps do
{
  #pragma omp parallel for private(i,l_x)
  for each tetrahedron i do
  {
    l_x=4*i;
     //Main computation
    Y[i] =
           A[l_x]*(X[I[l_x]]-X[i])+
           A[l_x+1]*(X[I[l_x+1]]-X[i])+
           A[l_x+2]*(X[I[l_x+2]]-X[i])+
           A[l_x+3]*(X[I[l_x+3]]-X[i]);
  }
  //Prepare for new timestep
  Swap(X,Y);
}
```

Figure 4: Pseudocode for the simple multicore implementation.

equivalent of the GPU kernels described in Section 2. Both steps are executed in the same *parallel* region, although the compute step is called multiple times. Since it uses the *parallel* pragma instead of the *parallel for* pragma, the *chunk-size*, i.e. the number of tetrahedra per thread must be computed explicitly. Note that this calculation is omitted in Figure 5.

4. Experimental Setup

Since the finite volume computation kernel is memory bound, its performance is primarily determined by the speed at which data is delivered to the processors. Thus, memory bandwidth sets an upper bound on the performance. However, only the minimum amount of data to be transferred to and from the device's global memory is known a priori. It amounts to reading 56 bytes and writing 8 bytes per tetrahedron, as discussed in Section 1. The actual amount of data to be loaded depends on the effectiveness of the caching. Moreover, there can be stalls due to global memory accesses. Thus, the actual data transfer rate may be significantly lower than the theoretical memory bandwidth.

To perform an experimental investigation of this effect, we construct instances that contain only disjoint blocks of b tetrahedra, i.e. each tetrahedron is only connected to neighbors only within its own block. The neighbors are chosen at random. For such instances the connectivity matrix, i.e. a binary matrix C where $C_{ij} = 1$ iff tetrahedron i is connected to tetrahedron j, has a block diagonal form. Thus, assuming the computation of all tetrahedra within one block is performed together, only b elements of x must be accessed during the computation of one block. Clearly, the minimum block size is 5. We measure the performance of our code for a large number of exponentially increasing block sizes on the K20 in order to understand the relationship between block

```
#pragma omp parallel private(i,l_x,loop)
{
  //Initialization
  my_tid   =omp_get_thread_num();
  double* Loc_A=malloc(chunksize*4*sizeof(double));
  int*    Loc_I=malloc(chunksize*4*sizeof(int));
  memcpy (Loc_A, A+(4*chunksize*my_tid),
                  4*chunksize*sizeof(double));
  memcpy (Loc_I, I+(4*chunksize*my_tid),
                  4*chunksize*sizeof(int));
  for #timesteps do
  {
    i=my_tid*chunksize;
    for (loop=0; loop<chunksize; loop++ )
    {
      l_x=4*loop;
      //Main computation
      Y[i] =
             Loc_A[l_x]*(X[Loc_I[l_x]]-X[i])+
             Loc_A[l_x+1]*(X[Loc_I[l_x+1]]-X[i])+
             Loc_A[l_x+2]*(X[Loc_I[l_x+2]]-X[i])+
             Loc_A[l_x+3]*(X[Loc_I[l_x+3]]-X[i]);
      i++;
    }
    //Prepare for new timestep
    #pragma omp barrier
    #pragma omp master
      Swap(X,Y);
    #pragma omp barrier
  }
}
```

Figure 5: Pseudocode for the full multicore implementation. Note that *chunksize* for each thread is obtained by dividing the tetrahedra evenly among the threads.

size and performance. In total, 100 such instances are generated for each experiment, although fewer datapoints are shown in our figures for the sake of clarity. Unless otherwise noted, n, i.e. the number of tetrahedra will be $1,000,000$ in these experiments, and the largest value of b among the test instances will be $500,000$.

In addition to the constructed instances, we use four real-world meshes to assess the realistically achievable GPU performance. Mesh 1, labeled "cardiac", is a mesh associated with a cardiac simulation. It has $600,791$ tetrahedra. The other three meshes are taken from [5]. Mesh 2, labeled "torso" is a 3D mesh of a torso and consists of $1,082,723$ tetrahedra. Meshes 3 and 4 are both from aeronautics applications having $222,414$ and $240,122$ tetrahedra respectively. They are labeled "blunt fin" and "F117".

All our GPU experiments are run on a Nvidia K20m Kepler GPU with ECC turned off. Their codes are compiled using nvcc 5.0. The CPU experiments are performed on dual Intel E5-2670 processors. CPU threads are pinned on physical cores using the *explicit KMP_AFFINITY* settings. This is necessary in order to obtain reproducible performance results. For cores residing on the same physical CPU socket, communication between them utilizes the L3 cache and is thus extremely fast. On the other hand, communication between the two physical sockets using Quick path interconnect is much slower at 32 GB/s [6]. The CPU codes are compiled using the Intel ICC 13.1.1 compiler. The processors are set to not use hyperthreading.

In order to obtain reliable performance measurements, 100 time steps are executed. In practice, only the first time step shows significantly lower performance, while all subsequent steps take approximately the same amount of time. This is due to the significant overhead of launching a CUDA kernel, which was measured to be 60 microseconds in [7]. At the end of each time step, the vectors x and y are swapped. Because we compute Eq. (1) for every tetrahedron in the mesh in every time step, performance values can be calculated as $n * 100 * 11/runtime$, since 11 FLOPS are performed for each tetrahedron. For the GPU experiments, we do not consider the time necessary for transferring data to and from main memory to device memory.

For the real-world meshes, we measure performance relative to block size in the same way as in the constructed instances. We use the graph partitioner METIS [8] and the hypergraph partitioner PaToH [9, 10] in order to obtain a block structured instance of the desired block size b. This amounts to a renumbering of the tetrahedra in such a way that every b consecutive tetrahedra have maximum connectivity among each other and minimum connectivity to tetrahedra in other blocks. However, since it is impossible to obtain arbitrary partitionings of real-world meshes, the results only approximate the desired block structure. Partitioning to desired block sizes smaller than 20 resulted in very weak approximations and significantly lower performance. Therefore, all the subsequent plots that concern the real-world meshes start from $b = 20$. The non-reordered instances are represented by the last data point in each curve, i.e. they have $b = n$.

5. Experimental Results on the GPU

In the first set of experiments, we investigate the performance of our kernels on the constructed instances of varying block size. We use four different numbers of the *threads-per-block* setting, i.e. the number of threads that are run together on a multiprocessor of the GPU. Note that this number is completely unrelated to the (instance or mesh) block size b, despite the similar name. Results for the shared memory based Kernel 1 are shown in Figure 6. Clearly, block size b is the main determinant of performance, while the thread block size has only a minor influence. For small values of b, all four curves show relatively constant performance. With increasing b, it drops smoothly before reaching a second plateau. For very large values of b, all four graphs drop sharply to a performance

of about 5 GFLOPS. Memory bandwidth can be assumed to account for the first plateau, and L2 cache bandwidth for the second. The low performance for very large values of b, i.e. on unordered instances, is a result of global memory latency. This is discussed in detail in Section 7.

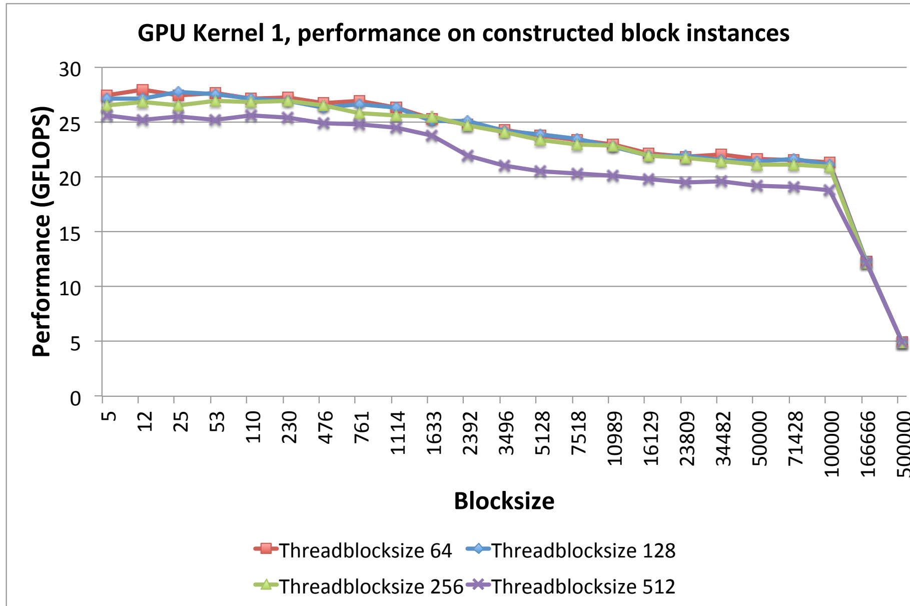

Figure 6: GPU performance on constructed instances as a function of the block size b and thread block size using Kernel 1.

The difference between the thread block sizes of 64, 128, and 256 is barely visible. The reason for these small differences lies in the *__synchthreads()* instruction, which causes all threads in a thread block to stall until loads to shared memory have been performed. Naturally, this synchronization cost leads to longer delays for larger thread blocks sizes, and therefore lower performance overall. Thus, a thread block size of 64 is always preferable for this kernel.

Next, we investigate the performance of our alternative read-only cache based Kernel 2. Results are shown in Figure 7. Here, the larger thread block sizes tend to provide better performance, while thread block size 64 behaves in a very different manner, showing comparatively lower performance for small b and higher performance than the alternatives for larger values of b. Figure 7 also includes Kernel 1 performance in order to illustrate the difference between the two kernels. Kernel 1 needs to copy data into the shared memory, which makes it slightly slower than Kernel 2 and limits its occupancy. For larger b, contention for the extremely limited read-only cache leads to significantly lower performance of Kernel 2. See Section 7 for details on this. When memory latency becomes the dominant factor, performance differences between the ker-

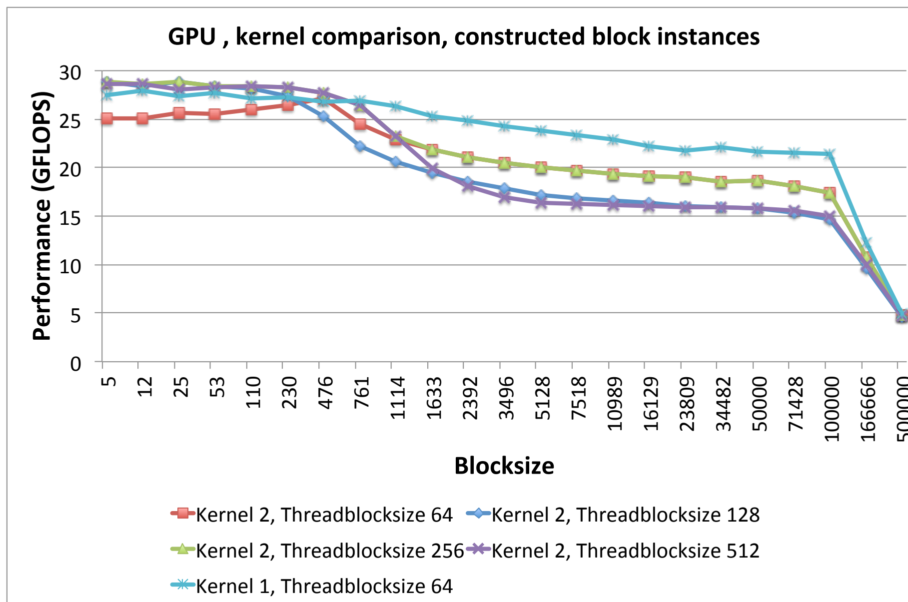

Figure 7: GPU performance on constructed instances as a function of the block size b and thread block size using Kernel 2. The performance of Kernel 1 is given for comparison.

nels even out. In conclusion, running Kernel 2 with 64 threads per block is always inferior to Kernel 1. Kernel 1 using 64 threads per block delivers the best performance overall while Kernel 2 using 512 threads is faster for small b. Therefore, in the following we will always use Kernel 1 with 64 and Kernel 2 with 512 threads per block.

In a second set of experiments, we study the performance of the real-world meshes. Figure 8 shows the performance for the instances created from Mesh 1 using METIS partitioning with varying desired b values. We see that even though the instances are quite different from the constructed ones, the performance curves are similar. Clearly, in case of very small b, Kernel 2 again shows slightly higher performance, while otherwise Kernel 1 is superior.

To illustrate the difference between the performance on both types of instances, we show a direct comparison for Mesh 2 in Figure 9. We chose Mesh 2 for the comparison because its number of tetrahedra is almost identical to that of the constructed instances. Clearly, performance is determined primarily by the kernel being used. The type of instance is clearly secondary. For small values of b, the constructed instances run faster because it is impossible to partition the meshes to the same quality. For larger b however, the partitioner manages to approximate the desired block structure closely, resulting in very similar performance for real-world and constructed instances. In fact, the placement of the indices $\mathcal{I}$ is less random in some of the partitioned real-world instances,

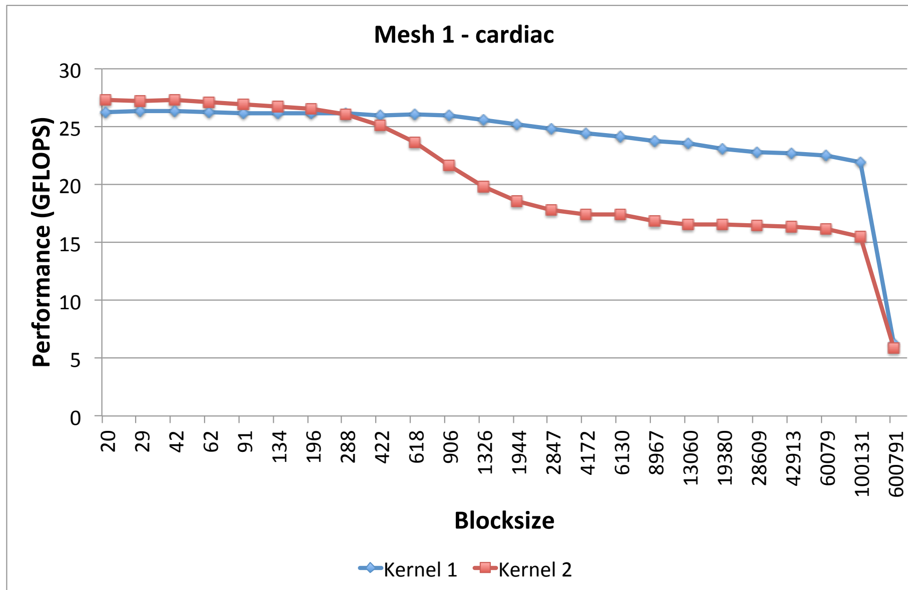

Figure 8: GPU performance on Mesh 1.

resulting in higher cache hit rates and thus higher performance. A similar effect is visible more clearly in figures 10 and 11.

Meshes 3 and 4 show noticeable differences for moderate and larger b values there. Both meshes only shows a small drop in performance for very large b due to memory latency. The reason for this is that both meshes start out as relatively well-ordered, i.e. having a small b. Since the partitioner does not place entries randomly within the blocks it generates, they keep this ordered structure while being partitioned into large blocks and therefore continue to resemble the well ordered instances. Meshes 1 and 2 on the other hand are originally unordered, and thus after being partitioned to large block sizes their entries are distributed in a sufficiently random manner, and thereby harder to cache effectively.

In a final set of experiments, we measure the influence of the quality of the partitioning. To this end, we compare previous results on the real-world instances obtained via METIS partitioning to results on instances generated using PaToH instead. In general, METIS tends to be very fast, but it often produces partitions of inferior quality compared to alternatives such as PaToH. For tetrahedral meshes, the graph partitioning approach used in METIS attempts to minimize the number of connections between tetrahedra in different blocks. PaToH on the other hand approximately minimizes the number of tetrahedra having at least one connection to a different block. However, the difference in performance between instances created using either partitioner is very small, as shown in Figure 12. In fact, the difference is hardly visible in the figure.

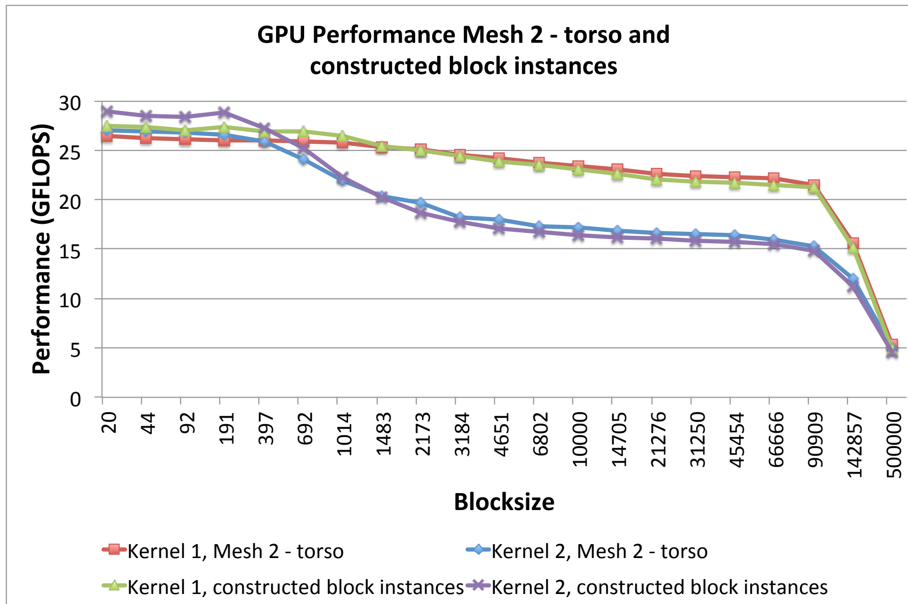

Figure 9: GPU performance on Mesh 2 compared to constructed instances.

It could stem from variations in the running time measurements alone, and is therefore not significant. Other instances showed essentially the same behavior. Thus, we conclude that small differences in the quality of the partitioning are not relevant, and it is preferable to use a fast partitioner. However, partitioning remains essential for obtaining an acceptable blocksize and thus performance. For example, as shown in Figure 8, both kernels show very low performance on the unpartitioned Mesh 1 with a blocksize b of 600791. However, for Kernel 1, at $b = 100131$ performance is already acceptable and smaller blocksizes only lead to marginal increase in performance. Therefore, the presence or absence of partitioning has a huge influence on performance, and unless the desired number of timesteps is very low, partitioning will pay off. However, the actual quality or desired blocksize does not affect performance strongly, as lower quality partitionings and higher block sizes have limited impact in this setting.

6. Experimental Results on the CPU

We study multicore CPU performance by running our implementations using between one and eight threads on a single socket, and 16 cores on two sockets. Results using randomly generated instance equivalent to those used for the GPU experiments are shown in Figure 13. Using a single socket, both implementation options described in Section 3 have identical and very stable performance. Therefore, they are merged in the figure. Performance doubles

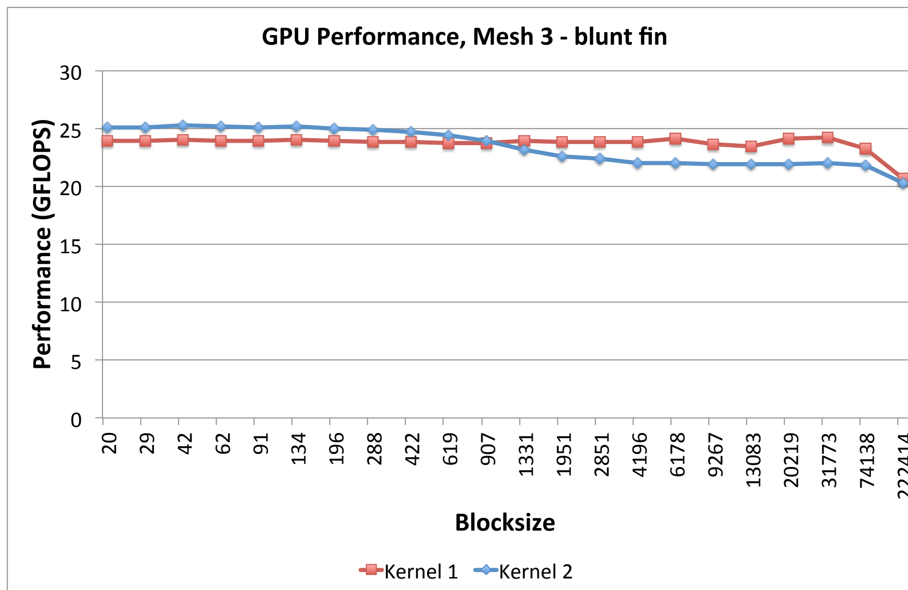

Figure 10: GPU performance on Meshes 3.

when going from one to two cores, and it triples for four cores. Using eight cores offer no improvement on small blocksizes and only a marginal improvement on medium blocksizes. On large block sizes however performance essentially doubles compared to four cores.

When using 16 cores and thus two sockets, performance of the implementations differs significantly. The simple implementation using the *parallel for* pragma does not scale and performs slightly worse than it does on eight cores, while the full implementation exhibits more than twice the performance. Therefore, in the following we restrict our attention to the full implementation.

In Section 5, we noticed that performance is closely related to instance and block size, but not to the specific instance. Therefore, we do not repeat the experiments on Meshes 1 through 4. Instead, we vary instance size and measure the resulting performance. Figure 14 shows performance for an instance size of $500,000$. Clearly, when using only one socket, performance results are essentially identical to those in Figure 13. For two sockets however, we attain up to 35 GFLOPS which is due to the fact that the entire work set fits in the 2×20MB L3 cache of the two CPUs. See Section 8 for a more detailed explanation of this effect. On the other hand, for larger instances, performance is significantly lower at about 11.5 GFLOPS for an instance size of $5,000,000$ tetrahedra, as shown in Figure 15.

Performance on the multicore CPUs differs from GPU performance in several ways. First, the dual CPUs provide a significantly lower level of maximum performance. Increasing the blocksize still results in three relatively flat levels

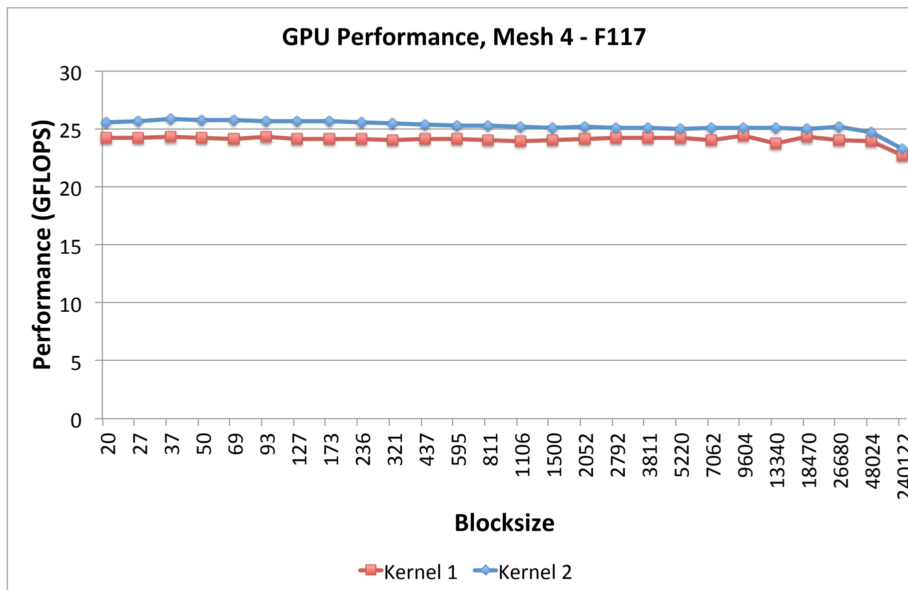

Figure 11: GPU performance on Meshes 4.

of performance, but the difference between these is much smaller than on the GPU. Using more than four cores, the first level disappears completely, as shown in Figure 13. Using 16 cores, the performance for large blocksizes is about two times higher than that of the GPU.

7. Performance modeling for the GPU

We have seen that the performance profiles of our kernels show a distinctive shape with respect to b for all test instances. The shape can be partially explained based on the characteristics of the hardware. To do so, we use the constructed instances and profile the accesses to L2 cache, as well as the L2 cache misses, i.e. the accesses to the device's global memory. Figures 16 and 17 show the results for kernels 1 and 2 respectively, using the standard thread block sizes of 64 and 512.

Clearly, memory bandwidth sets an upper bound on the performance, as observed in Figures 6 and 7. Consequently, even though the number of L2 accesses rises with increasing b, performance remains fairly constant up to $b \approx 200$. The maximum performance attained was 29.42 GFLOPS, which equals 82.3% of the theoretical maximum. Since the K20 has a theoretical memory bandwidth of 208 GB/s, its theoretical maximum performance is 35.75 GFLOPS because every tetrahedron requires reading 56 bytes and writing 8 bytes in order to perform 11 FLOPS. Thus, a performance of 29.42 GFLOPS implies a throughput of 171.17 GB/s.

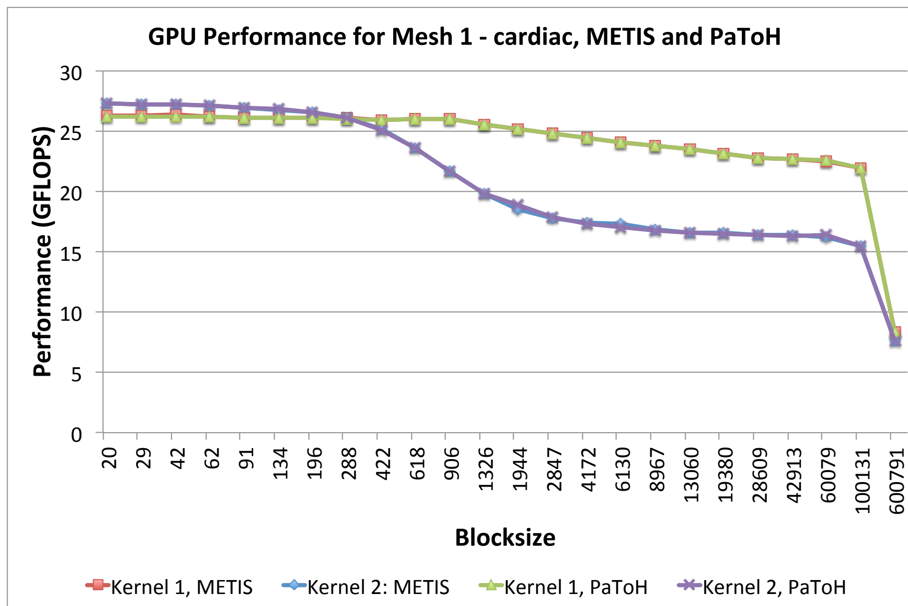

Figure 12: GPU performance for Mesh 1 using METIS and PaToH partitioning.

In [11], a vectorized memory benchmark attained 88% of the theoretical maximum on a K20x. For the K20, this would amount to 183 GB/s. However, our computation has to access multiple arrays and cannot benefit from all the optimizations employed there. We thus assume that Kernel 2 essentially exhausts the practically available bandwidth. Due to the copying into shared memory and the subsequent thread synchronization, Kernel 1 is slightly less efficient, but still bound by memory bandwidth.

The computation itself takes up an almost insignificant amount of time due to the fact that the K20 possesses 13 SMs, each possessing 64 double precision execution units running at 706 MHz and thus allowing $13*64*706$ million, which is 587.392 billion operations per second. Since the multiplication and addition of each of the four partial results are performed in a single operation, performing the 11 FLOPS per tetrahedron costs only 8 operations. Thus, computation runs at an effective $587.392 * \frac{11}{8} = 807.664$ GFLOPS which means that it is about 25 times faster than the memory bandwidth based limit. Thus, even if computation is interleaved with communication imperfectly, it has no meaningful influence on our results.

In any case, the full memory bandwidth can only be exploited if every access transfers *useful* data only. By useful, we mean data that is either used immediately, or can be stored in the next higher level of the memory hierarchy until it is used. Now, due to irregular data access, requesting a single entry of x will cause the device to transfer 32 bytes of data, i.e. 4 consecutive entries. If only

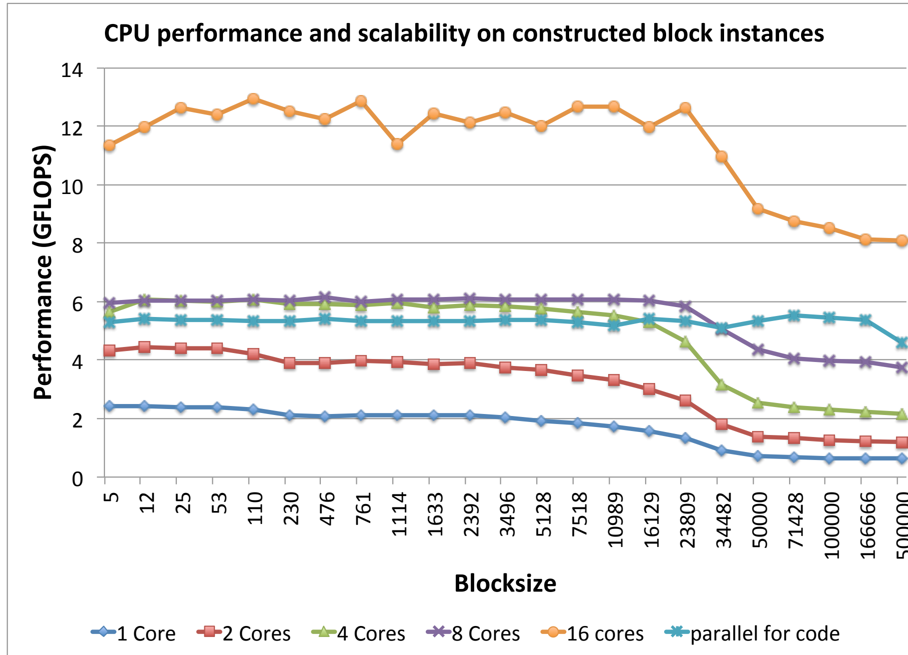

Figure 13: Performance of the CPU implementations on constructed instances.

one of these is currently needed, 24 bytes have been transferred in vain unless there is sufficient cache to store the additional data. In addition, due to varying cache or register reuse, the number of x entries to be transferred is not fixed. Under ideal circumstances, i.e. perfect caching and enough read-only cache to store a number of entries of x equal to b for all active threads, only one amortized entry of x per tetrahedron must be read. To see this, consider that for the first tetrahedron in a block, all b entries of x have been read due to caching. But then, for the remaining $b-1$ tetrahedra in the block, no further entries of x are required because they already reside in the cache. Since x is accessed contiguously in this case, all 32 bytes of data in a memory access can be used. Thus for x the amortized minimum bandwidth requirement per tetrahedron is 8 bytes or one fourth of a memory access.

If b is very large compared to the size of the read-only cache, $x(i)$ is still accessed contiguously during the computation of tetrahedron i. However, the four off-diagonal entries of x do not reside in read-only cache and must be fetched from L2. Since these accesses will most likely not be contiguous, every one of them will require a 32 byte memory access of which only 8 bytes will be used. Thus, in the worst case, each tetrahedron requires 4.25 accesses to L2 cache for the five entries of x alone.

Furthermore, A and $\mathcal{I}$ are accessed contiguously and thus they require 1.5 amortized memory accesses since their entries amount to 48 bytes. Thus, the

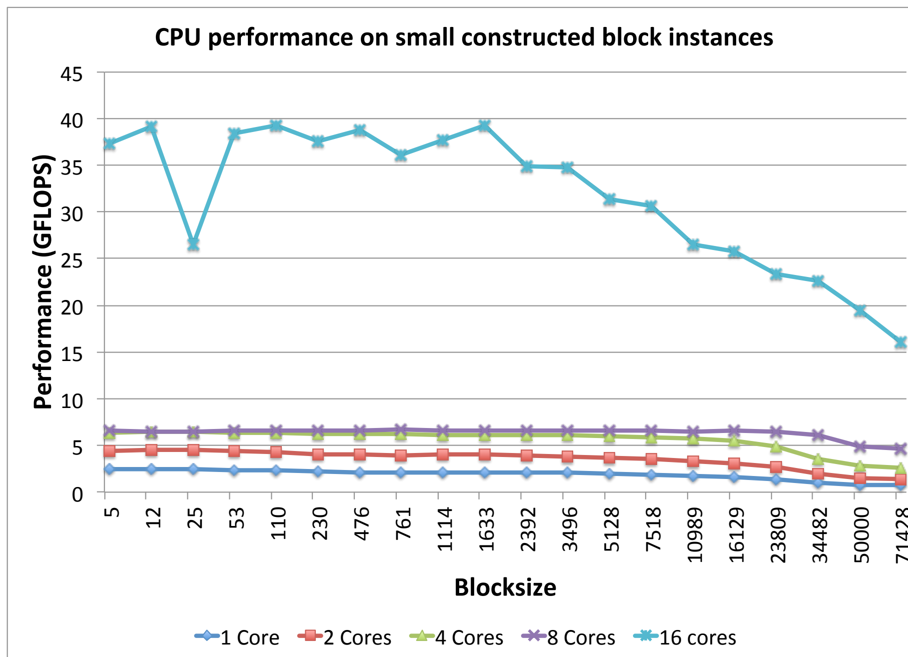

Figure 14: CPU Performance on small constructed instances.

absolute minimum number of amortized memory accesses is 1.75. In Figure 16, this happens for $b = 8$ because in that case, exactly two cache lines of x entries per block will be read and used. Since the thread block size is a multiple of 8 and thus aligned with b, no block will be spread among multiple thread blocks and thereby possibly among multiple SMs. Thus, every entry of x will be transferred exactly once.

On the other hand, the upper bound is 5.75 since the maximum number of x accesses per tetrahedron is 4.25. In addition, 8 bytes, i.e. 0.25 cache lines per tetrahedron must be written back to memory which implies a final memory transfer requirement of 192 bytes per row and thus a computational intensity of 0.057. Since the K20 has a theoretical memory bandwidth of 208 GB/s, its performance in that case would be capped at 11.91 GFLOPS. However, due to memory latency the worst case performance is even lower.

This becomes an issue when b rises to a value where L2 cache is no longer sufficient to cache x. Since the device's $1,280$KB of L2 cache can contain $160,000$ double values, this point is reached at $b = 80,000$, as shown in figures 16 and 17. If b is larger than that, tetrahedra from up to two mesh blocks can be processed concurrently with some belonging to the end of the first block and others belonging to the start of the second block. Thus, to cache all potentially relevant entries of x, $160,000$ entries need to be stored in L2 cache simultaneously. Due to the large number of active threads, the number of L2 misses and

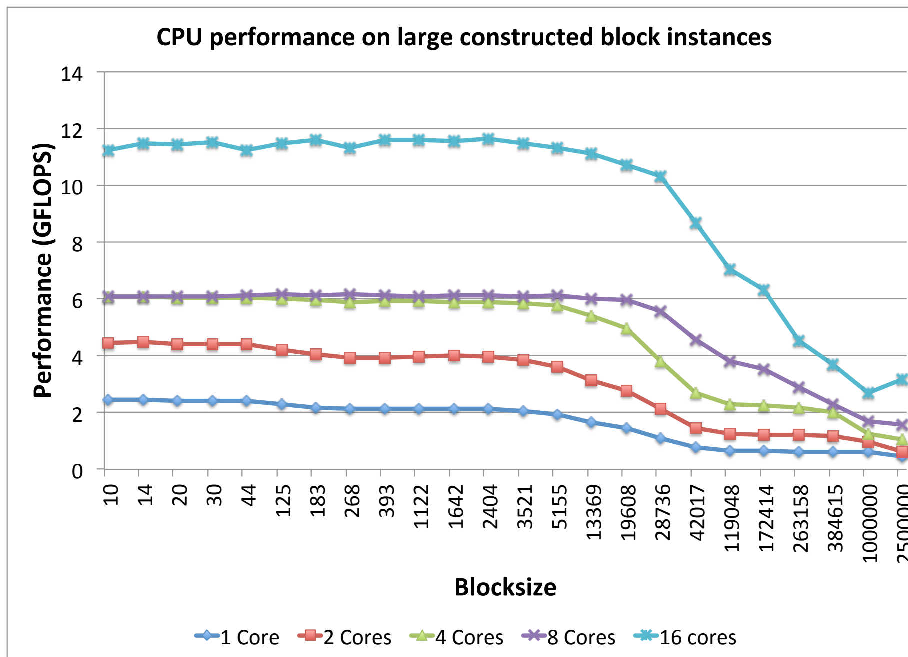

Figure 15: CPU Performance on large constructed instances.

thus main memory accesses start to increase at a slightly lower value of b, especially for Kernel 2. The exact value depends on the caching policy and on how much space is used for caching A and $\mathcal{I}$. Note that before this point, L2 caches almost perfectly. The number of misses remains constant at the minimum of 1.75 per tetrahedron. Since memory is accessed contiguously, these accesses are very fast, as discussed above. On the other hand, once the number of memory accesses starts rising at $b = 80,000$, random memory accesses ensue and performance decreases dramatically due to the high memory latency of about 1000 cycles. Thus, $16 \times b$ bytes of L2 cache are required for acceptable performance on instances of block size b, which amounts to $1,280$KB for a block size b of $80,000$.

Within the region of b between 200 and $80,000$, performance is determined by the bandwidth of the L2 cache. Unlike the first level of caches, it serves all 13 SMs of the device and it caches all memory accesses, making it a potential bottleneck. The theoretical L2 bandwidth of the K20m is 542 GB/s, due to a

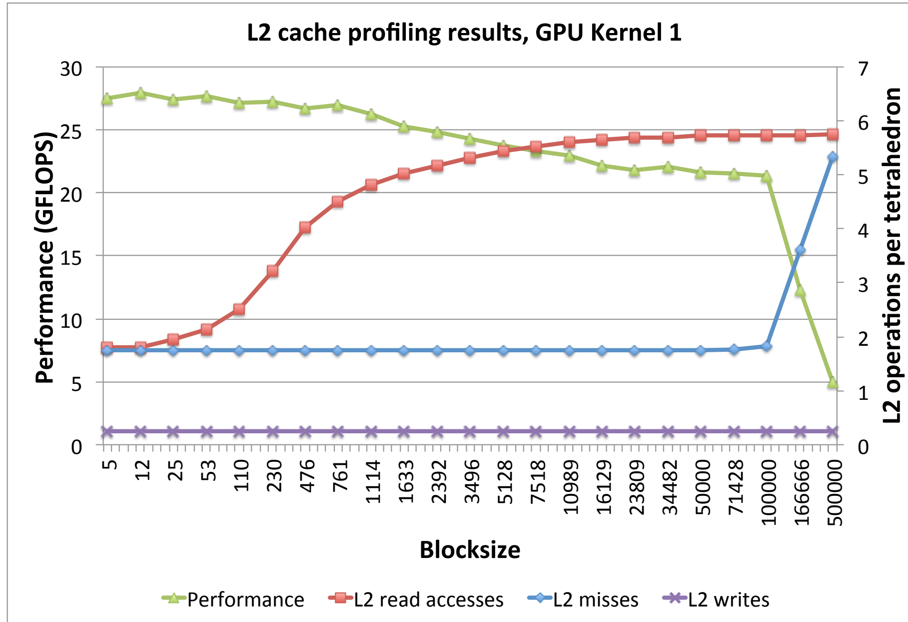

Figure 16: GPU performance, L2 read accesses, L2 write accesses, and memory accesses by block size b.

768 Byte per cycle capacity at 706 MHz[1]. However, it can be assumed that its practically available bandwidth is significantly lower, although this is difficult to verify since the exact caching policy is not made publicly known by the manufacturer.

To study the connection between performance and L2 bandwidth in that region, we compute the L2 bandwidth that would be required to transfer 32 bytes per L2 access shown in Figure 16. Multiplying the sum of read and write accesses by 32 yields the amount of data transfer per tetrahedron in bytes. Performance in FLOPS divided by 11 yields the number of tetrahedra processed per second. The product of these numbers is the required L2 bandwidth. The maximum required L2 bandwidth computed in this way is 394.1 GB/s at $b = 13333$. For this block size we measured 5.62 read and 0.25 write accesses, and a performance of 23.08 GFLOPS.

For Kernel 1, in the region of b between $1,000$ and $80,000$, given an L2 bandwidth of 394.1 GB/s, we found that performance can be reproduced from the actual number of L2 accesses with less than 3% error. While this data does

[1]In [1] the L2 cache bandwidth per cycle of the Kepler device is declared to be twice that of Fermi, i.e. 768 byte per cycle. At 706 MHz clock rate, this amounts to 542 GB/s. Communication with Nvidia confirmed an L2 bandwidth in the order of twice the memory bandwidth.

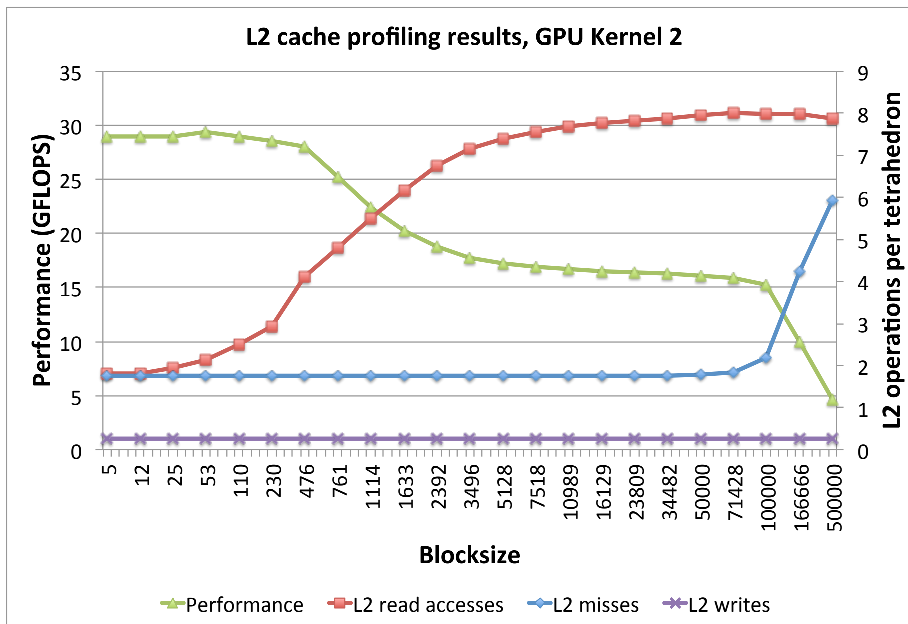

Figure 17: GPU performance, L2 read accesses, L2 write accesses, and memory accesses by block size b.

not preclude the possibility that the practical peak for other applications might be a bit higher, we can assume that the computation is L2 bandwidth bound in this region. This also explains why the curves for 64, 128, and 256 threads per block overlap in this section of Figure 6.

For Kernel 2, the situation is quite similar in this region. However, due to contention for the limited read-only cache, a significant amount of overfetching from L2 occurs because elements of x are evicted in order to make space for new elements of A and $\mathcal{I}$. Thus, instead of the Kernel 1 maximum of 5.75 read accesses per tetrahedron, up to 8 accesses to L2 occur. Because the computation is L2 bandwidth bound here, performance is significantly lower. However, assuming a realistic L2 bandwidth of 394.1 GB/s, this performance can again be reproduced from the heightened number of L2 accesses with less than 3% error.

From a practical point of view, these results can be used to optimize finite volume computations on the GPU. Obviously, unless the input is already well-ordered, using a partitioner to obtain a good reordering[2] of the mesh is necessary for competitive performance. Since performance is bound by memory

[2]Traditional renumbering methods such as the Cuthill-McKee algorithm [12] give similar results. However, they do not allow selecting a desired block size.

bandwidth, it is not helpful to attempt to create very small blocks. Instead, somewhat larger blocks give the partitioner more freedom in case of difficult meshes without compromising performance. Due to alignment effects, matching b to thread block size or multiples thereof is preferable. Thus a size of 64 or 128 represents a good choice for b. The higher quality partitioning provided by PaToH does not yield any added benefits. For a GPU with different memory and L2 bandwidth, an equivalent suggestion can be found as follows.

Let BW_M be the memory bandwidth and BW_{L2} the L2 cache bandwidth. The memory limit performance is $BW_M * 11/64$ GFLOPS. For the L2 bound, let a be the number of L2 accesses per tetrahedron. Now, the L2 limit performance is $BW_{L2} * 11/32a$ GFLOPS. Figure 16 shows the connection between L2 accesses and b. The smallest b at which the L2 limit performance becomes smaller than the memory limit performance is the largest b that can be used for partitioning. Of course, depending on the GPU, the two performance limits do not necessarily intersect. To determine a easily, consider Eq. (2). It defines the above intersection:

$$\frac{11BW_{L2}}{32a} = \frac{11BW_M}{64} \Leftrightarrow a = \frac{2BW_{L2}}{BW_M} \tag{2}$$

Thus we obtain a as the rate of L2 accesses at which the kernel stops to be bound by memory bandwidth and starts to be bound by L2 bandwidth. The value of a for a given value of b can be obtained from Figure 17. By solving Eq. (2) for BW_{L2}, we can compute the required L2 bandwidth relative to the memory bandwidth, assuming the number of L2 misses is constant, i.e. an L2 cache of unlimited size.

The intersection point occurs at around $b = 500$ for Kernel 1 and at $b = 700$ for Kernel 2. However, the performance curves do not drop sharply. Instead, between $b = 200$ and $b = 1000$, both curves drop smoothly, showing performance that is slightly below either memory or L2 bandwidth limit.

The first level of caching, i.e. the shared memory and the read-only cache do not constitute bottlenecks. Their bandwidth is in the order of 1 TB/s [4], and this bandwidth is available to each SM separately. On the other hand, their sizes can affect maximum performance here. A larger read-only cache would decrease contention in Kernel 2, raising its L2 bound performance towards that of Kernel 1. For both kernels the number of L2 accesses would rise more slowly, thus increasing the L2 bandwidth bound performance. A larger shared memory would be helpful to raise the performance of Kernel 1 towards that of Kernel 2 for small values of b. It has no effect on Kernel 2.

8. Performance modeling for the CPU

In order to model the attainable CPU performance, we first consider the memory bandwidth. Each Sandy Bridge CPU has a theoretical memory bandwidth of 51.2 GB/s giving a total of 102.4 GB/s on our test system. This

implies a theoretical maximum performance of 17.6 GFLOPS. In [13], a benchmark value of 88.8 GB/s for the practical maximum was measured. The maximum performance obtained in Figure 13 from Section 6 is 13.5 GFLOPS, i.e. about 75% of the theoretical maximum, which implies that the CPU is about 7 percentage points less efficient than the GPU.

However, consulting Figure 14 we note a maximum performance of 39.5 GFLOPS when using two sockets, which is above the limit set by the Roofline model. The reason for this is that in this case the entire instance fits in the combined L3 cache of the processors. Recall that the small constructed instances have $500,000$ tetrahedra which cost 56 bytes each, for a total of 28 MB. The vector x must be cached at least partially on both processors, but this implies an upper limit of 32 MB, which is still well below the 40 MB of combined cache. In [13], the L3 cache bandwidth was measured at 426.8 GB/s, which enables a maximum performance of about 73 GFLOPS. However, assuming the entire x vector must be communicated along the QuickPath interconnect, performance is limited to 44 GFLOPS. This number is based on the fact that for each 11 FLOPS, one value of x must be communicated along the processor. At 32 GB/s [6], 4 billion entries of x can be communicated per second, which implies the limit above since ideally only one x value per tetrahedron is transferred on average. While it is possible to modify the code in such a way to alleviate this restriction, we did not do so since this is not the focus of our study.

The speed of the QuickPath interconnect also plays an important role in understanding the weak 16 core performance of the *parallel for* implementation. In the worst case, all data is stored in the memory of socket 0, and all memory accesses on socket 1 are limited to the interconnect speed. However, since this would also imply that all memory accesses are handled by the memory controller of socket 0 with a memory bandwidth of 51.2 GB/s, i.e. less than half of the interconnect speed, an even lower limit applies. Interestingly, the *parallel for* implementation is still limited to 5.5 GFLOPS, which is the theoretical maximum that applies if all data was moved between the processors. Changing the *parallel for* schedule from `static` to either `dynamic` or `guided` does not improve the performance.

For a single socket, we reliably obtain a performance of 6 GFLOPS, independent of the number of tetrahedra. This implies a memory transfer speed of 34.86 GB/s, which amounts to 68% of the theoretical maximum and 79% of the benchmarked value of 44GB/s in [13]. This means that for very small blocksizes, in addition to having a higher theoretical peak, the GPU also has a significantly higher efficiency.

The available memory bandwidth of the Sandy Bridge CPU depends on the number of cores used [13]. However, the relation is not linear. It starts at about 18 GB/s for one core, 32 GB/s for two, and 44 for four cores and above. Consequently, performance for small values of b scales from 2.4 GFLOPS using one core to 4.4 GFLOPS using two cores, and 6 GFLOPS using four or eight cores. Clearly, for small values of b performance is purely memory bound.

With respect to blocksize, we use a model similar to that for the GPU. The main difference is that the different levels of cache have very similar bandwidth.

Furthermore, the cacheline size is 64 bytes instead of 32, which changes the required cache bandwidth in case of larger blocksizes. Since the computation requires at least 56 bytes per tetrahedron, an amortized minimum of 0.875 cache lines have to be read. In case of high irregularity, up to four additional cache lines have to be read, as described in Section 7. This gives us a maximum of 4.875 cache lines and thus 312 read bytes plus 8 written bytes per tetrahedron. In that case, each socket would be limited to a theoretical 1.76 GFLOPS. However, this level of performance can only be reached when the blocksize is large compared to the cache. Due to the large L3 cache on the CPU, this only happens for very large instances, unlike on the GPU. In Figure 15, the minimum performance reached is 1.575 GFLOPS for 8 cores and twice as much for 16 cores, indicating that indeed memory bandwidth is a limiting factor on the CPU when b is very large, unlike on the GPU where memory latency dominates.

In addition to memory, L3 cache bandwidth can be a limiting factor as well, similar to the L2 bottleneck of the GPU. Revisiting Eq. (2), we obtain for the CPU:

$$\frac{11BW_{L3}}{64a} = \frac{11BW_M}{64} \Leftrightarrow a = \frac{BW_{L3}}{BW_M} \tag{3}$$

This means that the number of cache lines used per tetrahedron cannot be higher than the ratio between L3 cache and memory bandwidth before L3 bandwidth becomes a bottleneck. In [13], it was measured at 30.22 GB/s for one core, and memory bandwidth was measured at 17.16 GB/s, suggesting that this is indeed the case for larger memory bandwidth.

At 320 bytes per tetrahedron, L3 bandwidth limits each core to about 1 GFLOPS or to 2 GFLPOS at 160 bytes per tetrahedron which seems to be the performance for blocksizes between 500 to 2000 in Figure 13. Judging from the expected cache misses shown in Figure 16, for up to 4 cores performance seems to be indeed bound by cache bandwidth, similar to the GPU. However, using 8 cores, ample cache bandwidth is available which leads to the memory bandwidth being the limiting factor.

For $1, 2$, and 4 cores, i.e. when cache bandwidth is a limit, we observe three distinct drops in performance. The first is at about $b = 100$. This is due to the fact that the register file only holds 144 floating point entries, and some will be occupied by values of A. Since the CPU has relatively low L1 bandwidth, irregular accesses to L1 cache cause a noticeable drop in performance. The second drop starts around $b = 4K$, which is the number of x entries cacheable in 32KB L1 data cache. The third drop is at $b = 32K$, which stems from 256KB L2 cache. Note that the transitions from higher to lower performance are smooth, since having slightly more data than cache means that the number of cache misses increases only marginally. In Figure 15, a fourth step appears. Because L3 cache is shared among the cores, its nominal capacity of 2.5 million x entries must be divided by the number of cores. Therefore, the effect of this last step varies by number of cores used. In case of 16 cores, twice the L3 cache

is available, resulting in a much higher performance on very large block sizes in Figure 15.

Thus, we can conclude that CPU performance is bound by memory bandwidth for very small or very large block sizes. Between those however, cache bandwidth limits performance, much like on the GPU. When using multiple sockets, performance is also subject to NUMA effects and at least partially limited by the speed of the QuickPath Interconnect. Unlike on the GPU, in our experiments for very large block sizes, performance is limited by memory bandwidth rather than latency.

9. A General Simplified Performance Model

In the following we present a generalized model for irregular application performance based on the above considerations. It is applicable for computations in which the average number of required data per operation is known beforehand. In addition, the model requires an upper bound on the data that would benefit from being cached akin to the block size b used in this paper.

In order to simplify the discussion, we use the word length w as the number of bytes a single variable in the computation occupies. In most scientific computations, these are double precision floating point values that have $w = 8$.

We have seen that performance in irregular applications such as the cell-centered finite volume computation depends largely on the ratio between effectively consumed bandwidth and available bandwidth at every level of the memory hierarchy. The effectively consumed bandwidth can be approximated as follows: for the irregular part of the data, we have to determine how many cache lines per word have to be transferred on average. Let a_i be this number for cache level i. Its inverse is known as useful cache line density. Let C_i be the size of the cache at level i in words, and let W be the size of the working set, i.e. the number of words that could be accessed at any given time and thus should ideally be cached. For $C_i > W$, assume $C_i = W$ since excess cache is effectively wasted. In our application, $W = b$. Furthermore, let CL_i be the cache line size of level i in words.

Now the cache hit rate at level i can be estimated by $\min(1, C_i/W)$. Cache hits at level i imply that cache line transfers from level $i + 1$ transmit CL_{i+1} useful words. Cache misses imply a transfer rate of one useful word per cache line. Thus, one word of irregular data consumes a bandwidth of $\min(1, C_i/W) + (1 - \min(1, C_i/W))CL_{i+1}$. This is a simplification which ignores the possibility of transferring more than one word on a cache miss. The regular data consumes bandwidth simply at a rate of one word per word of data. Thus assuming that we need R regular and U irregular data words per flop, the total number of words per flop on level i is therefore:

$$WpF_i = R + U(\min(1, C_i/W) + (1 - \min(1, C_i/W))CL_{i+1}) \quad (4)$$

The bottleneck performance P_i at level i is then $\frac{BW_i}{WpF_i w}$, assuming the bandwidth BW_i is given in bytes per second. By calculating $P_{min} = \min_i P_i$, we obtain an

Level	Size C_{i-1}	CL_i	Bandwidth (GB/s)
L1	140	8	35.31
L2	$4,000$	8	35.14
L3	$32,000$	8	30.22
Memory	$2,500,000$	8	17.16

Table 1: Memory hierarchy features of a single Sandy Bridge CPU core. Note that size refers to the size of the level above, since cache misses on the level above determine accesses to a given level.

estimate on the bottleneck performance of an irregular application in FLOPS. The full model is thus given by:

$$P_{min} = \min_i\left(\frac{BW_i}{w(R + U(\min(1, C_i/W) + (1 - \min(1, C_i/W))CL_{i+1})}\right)) \quad (5)$$

Note that the registers form a level 0 of "cache". Since they are generally very small, most irregular accesses to L1 cache will provide only little useful data. Thus, for irregular applications a high L1 cache bandwidth is needed to compensate for that.

For the finite volume computation given by Eq. 1, denoted below as FV, we have $R = \frac{8}{11}$, due to 56 read and 8 written bytes per tetrahedron, and $U = \frac{4}{11}$. However, since cache hits on x imply that no data needs to be read, at every cache level i we have:

$$WpF_i(FV) = \frac{8 + 4((1 - \min(1, C_i/W))CL_{i+1})}{11} \quad (6)$$

The last "cache level" i is the memory. Its bandwidth imposes a limit even for perfectly regular computations. The Roofline model [14] discusses this in detail. In irregular applications, the effectively consumed bandwidth is higher than the data transfers would imply, which renders the Roofline model inaccurate here.

We test our model by applying it to the finite volume computation on a single CPU core. Here, the memory hierarchy contains five levels. Table 1 summarizes the relevant values for each level. Note that all values are in words, i.e. 8 bytes. Cache sizes are vendor-supplied information, and do not account for the fact that caches are partially occupied by regular data. Bandwidth is based on benchmarks in [13] and given in GB/s.

By inserting these values into Eq. 5, we obtain predictions on the bottleneck bandwidth between the levels of the memory hierarchy. Table 2 shows the results for several different values of W. These values were chosen to align with cache sizes, which allows us to showcase drops in performance when exceeding the size of a given level of cache. Bottleneck performance values are given for the source of reads, i.e. L3 refers to the read bandwidth from L3 to L2 cache. Registers are therefore omitted here.

When comparing the minimum among the bottleneck performance values, i.e. the performance predicted by our model to the measured performance, we

	W				
Level	140	$4,000$	$32,000$	$500,000$	$2,500,000$
L1	6.07	1.24	1.21	1.21	1.21
L2	6.04	6.04	1.34	1.22	1.21
L3	5.19	5.19	5.19	1.09	1.04
Memory	2.95	2.95	2.95	2.95	2.95
Minimum	2.95	1.24	1.21	1.09	1.04
Measured	2.27	2.00	1.02	0.62	0.44

Table 2: Predicted bottleneck performance values at all levels of the Sandy Bridge memory hierarchy. Values are given in GFLOPS.

note that predicted values tend to be higher than actual measurements, which is to be expected since the model cannot capture all the complexities that limit performance in this computation. However, the prediction for $W = 4,000$ is noticeably lower than the measured performance, even though the model correctly predicts a drop in performance at that point. The most likely reason for overestimating the required L1 bandwidth lies in the fact that we do not consider irregular accesses that have more than one useful data word in the cache line.

In addition, memory latency can effectively slow down memory traffic further, which results in the low measured performance for $W = 2,500,000$. This cannot be captured in a simple performance model such as this, since latency can be hidden by parallel thread execution. A model able to capture this would have to be very application- and architecture-specific.

10. Related Work

Several papers [4, 15] study the architectural parameters of previous generations of GPUs. Using microbenchmarks, they measure hardware properties not supplied by the vendor. However, since not only numerical values but also properties such as cache policies are unknown, the applicability of this approach is limited. We do not rely on microbenchmarks for the GPU. Nevertheless, our performance measurements strongly indicate that the test code saturates memory and cache bandwidth.

Finite volume computations on the GPU have been studied in the context of various applications [16, 17, 18]. Such computations on tetrahedral meshes are closely related to sparse matrix-vector multiplication (SpMV), which is one of the most important operations in scientific computing. It has been studied extensively in the past, and high quality library implementations are readily available [19, 20]. Since the speed of such computations is limited by memory bandwidth rather than computational speed, GPUs are potentially well suited to this task because they usually offer significantly higher memory bandwidths than CPUs. Thus parallelization of SpMV on GPUs has seen significant research activity recently [3, 21, 22]. In [3], sparse matrix-vector multiplication with a

constant number of nonzeroes per row was studied. This is known as the ELLPACK format. Since less index data needs to be stored and transferred, it runs significantly faster than sparse matrix-vector multiplications using more traditional data formats such as compressed sparse rows (CSR). Furthermore, variants of the ELLPACK format have been applied successfully in general SpMV computations [23, 24, 25], rendering CSR obsolete on the GPU and on Xeon Phi accelerators [25, 26]. For such computations, the experiences gathered in this work are quite relevant due to their strong similarity. In fact, our code can be modified to perform ELLPACK SpMV computations. Doing so yielded approximately the same bandwidth, but lower FLOPS due to the lower computational intensity. By the same token, while our results are based on tetrahedral meshes with four neighbors per cell, they can be extended easily to other instances as long as the number of neighbors is constant except for boundary cells.

With respect to performance modeling, for SpMV and similar computations on multicore processors the Roofline model [14] is well established. It describes the maximum attainable performance as a function of processing capabilities, memory bandwidth, and the computational intensity of the problem. While the authors do suggest applying the model to the cache if it can hold the entire working set, they do not model the interplay of different layers of the memory hierarchy. Our model expands upon that by estimating the additional bandwidth consumed by irregular accesses.

11. Conclusion

We presented two different CUDA kernels for finite volume computations and investigated their performance on the K20 Kepler GPU. Kernel 2 attained 82.3% of the theoretical maximum performance, which is bounded by the global memory bandwidth. On Intel multicore CPUs, our code attained about 70% of the theoretical maximum.

We have identified cache bandwidth as a potential secondary bottleneck, which can limit the performance in addition to memory bandwidth. Although for unstructured tetrahedral meshes it can often be circumvented using proper reordering techniques, highly irregular applications will be bound by this limit and require coalescing techniques similar to Kernel 1. The size of the L2 cache can also limit performance when data access patterns are highly randomized. The CPU faces similar limitations, although it possesses a very large L3 cache.

The tested real-world applications closely follow our performance modeling, attaining 75% of the theoretical maximum performance and 93% of the realistic upper limit, thus showing that the device is well suited for this type of computation.

Finally, we generalized the performance modeling to provide a model for a large class of irregular applications. It takes the entire memory hierarchy and the possibility of wasting bandwidth due to overfetching into account, thereby allowing predictions of irregular application performance.

In future work we aim to improve our model by introducing better estimates for cache hits and misses, since these predictions are the main source of inaccuracy in the current model.

[1] NVIDIA Corporation, NVIDIA's next generation CUDA compute architecture: Kepler GK110, Tech. rep. (November 2012).

[2] D. K. R. Grimes, D. Young, Itpack users guide, Tech. Rep. CNA-150, Center for Numerical Analysis, University of Texas (Aug 1979).

[3] N. Bell, M. Garland, Efficient sparse matrix-vector multiplication on CUDA, NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation (December 2008).

[4] A. Schäfer, D. Fey, High performance stencil code algorithms for GPGPUs, Procedia Computer Science 4 (2011) 2027 – 2036, proceedings of the International Conference on Computational Science, ICCS.

[5] University of Maryland, CMSC 741, http://www.cs.umd.edu/class/fall2010/cmsc741/datasets.html (2010).

[6] Intel Corporation, An introduction to the Intel QuickPath interconnect, Tech. rep. (January 2009).

[7] V. Haydin, NVIDIA Tesla K20 Benchmark: Facts, Figures and Some Conclusions, http://elekslabs.com/2012/11/nvidia-tesla-k20-benchmark-facts.html (November 2012).

[8] G. Karypis, V. Kumar, Multilevel k-way hypergraph partitioning, VLSI Des. 11 (2000) 285–300.

[9] U. V. Çatalyürek, C. Aykanat, A hypergraph model for mapping repeated sparse matrix-vector product computations onto multicomputers, in: Proceedings of International Conference on High Performance Computing, 1995.

[10] Ü. V. Çatalyürek, C. Aykanat, A fine-grain hypergraph model for 2D decomposition of sparse matrices, in: Proceedings of 15th International Parallel and Distributed Processing Symposium (IPDPS), San Francisco, CA, 2001.

[11] F. Duguet, Kepler vs Xeon Phi : Nos mesures - et leur code source complet, http://www.hpcmagazine.fr/en-couverture/kepler-vs-xeon-phi-nos-mesures (June 2013).

[12] E. Cuthill, J. McKee, Reducing the bandwidth of sparse symmetric matrices, in: Proceedings of the 1969 24th national conference, ACM '69, ACM, New York, NY, USA, 1969, pp. 157–172. doi:10.1145/800195.805928. URL http://doi.acm.org/10.1145/800195.805928

[13] W. Zhang, W. Wei, X. Cai, Performance modeling of serial and parallel implementations of the fractional adams-bashforth-moulton method, Fractional Calculus & Applied Analysis 17 (2014) 617–637.

[14] S. Williams, A. Waterman, D. Patterson, Roofline: an insightful visual performance model for multicore architectures, Commun. ACM 52 (4) (2009) 65–76. doi:10.1145/1498765.1498785.
URL `http://doi.acm.org/10.1145/1498765.1498785`

[15] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, A. Moshovos, Demystifying GPU microarchitecture through microbenchmarking, in: Performance Analysis of Systems Software (ISPASS), 2010, pp. 235–246. doi:10.1109/ISPASS.2010.5452013.

[16] M. J. Castro, S. Ortega, M. de la Asunción, J. M. Mantas, J. M. Gallardo, GPU computing for shallow water flow simulation based on finite volume schemes, Comptes Rendus Mécanique 339 (2011) 165 – 184.

[17] B. Hamilton, C. J. Webb, Room acoustics modelling using GPU-accelerated finite difference and finite volume methods on a face-centered cubic grid, Proc. Digital Audio Effects (DAFx), Maynooth, Ireland.

[18] M. Long, D. He, Hydraulic erosion simulation using finite volume method on graphics processing unit, in: Information Engineering and Computer Science. ICIECS 2009., 2009, pp. 1–4. doi:10.1109/ICIECS.2009.5363863.

[19] S. Balay, J. Brown, , K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, H. Zhang, PETSc users manual, Tech. Rep. ANL-95/11 - Revision 3.4, Argonne National Laboratory (2013).

[20] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, K. S. Stanley, An overview of the Trilinos project, ACM Trans. Math. Softw. 31 (3) (2005) 397–423. doi:http://doi.acm.org/10.1145/1089014.1089021.

[21] J. W. Choi, A. Singh, R. W. Vuduc, Model-driven autotuning of sparse matrix-vector multiply on GPUs, SIGPLAN Not. 45 (5) (2010) 115–126. doi:10.1145/1837853.1693471.
URL `http://doi.acm.org/10.1145/1837853.1693471`

[22] X. Feng, H. Jin, R. Zheng, K. Hu, J. Zeng, Z. Shao, Optimization of sparse matrix-vector multiplication with variant CSR on GPUs, in: Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on, 2011, pp. 165–172. doi:10.1109/ICPADS.2011.91.

[23] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann, A. R. Bishop, Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation, CoRR abs/1112.5588.

[24] F. Vázquez, J. J. Fernández, E. M. Garzón, A new approach for sparse matrix vector product on NVIDIA GPUs, Concurr. Comput. : Pract. Exper. 23 (8) (2011) 815–826. doi:10.1002/cpe.1658.
URL http://dx.doi.org/10.1002/cpe.1658

[25] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. R. Bishop, A unified sparse matrix data format for modern processors with wide simd units, CoRR abs/1307.6209.

[26] A.-J. Yzelman, D. Roose, High-level strategies for parallel shared-memory sparse matrix-vector multiplication, Parallel and Distributed Systems, IEEE Transactions on 25 (1) (2014) 116–125. doi:10.1109/TPDS.2013.31.