A cross-vendor implementation of PopSift using SYCL

Mohammad Fadel Al Khafaji
University of Oslo
and Simula Research Laboratory
Oslo, Norway
mohafal@ifi.uio.no

Carsten Griwodz
University of Oslo
Oslo, Norway
griff@ifi.uio.no

Håkon Kvale Stensland
University of Oslo
and Simula Research Laboratory
Oslo, Norway
haakonks@simula.no

Abstract

Implementing the SIFT algorithm by Lowe [9], PopSift is an open-source CUDA implementation used to extract and describe keypoints in 2D images [5]. It is meant to be a faithful SIFT implementation and a stage of the photogrammetry pipeline that handles natural feature extraction of AliceVision Meshroom [2, 6], a 3D Computer Vision framework. It is used in a range from object recognition to 3D reconstruction. However, it is vendor-locked to Nvidia GPUs as it is written in CUDA, limiting its use and exposure. Therefore, reimplementing the application in SYCL [7] makes it available on a wider range of platforms. PopSift-SYCL is a portable, cross-vendor, and efficient port of the original PopSift, achieving both high speed and correctness. It can be compiled by both Intel's DPC++ compiler [4] and the open source AdaptiveCPP compiler [3, 1]. We present the stages of PopSift-SYCL and its performance.

CCS Concepts

• **Computing methodologies** → **Parallel programming languages**; **Image processing**; • **Computer systems organization** → *Single instruction, multiple data.*

Keywords

SIFT, SYCL, GPU computing, feature extraction, image processing, parallel programming, cross-vendor portability, computer vision

ACM Reference Format:
Mohammad Fadel Al Khafaji, Carsten Griwodz, and Håkon Kvale Stensland. 2026. A cross-vendor implementation of PopSift using SYCL. In *International Workshop on OpenCL and SYCL (IWOCL '26), May 06–08, 2026, Heilbronn, Germany*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3811257.3811261

1 Introduction

The Scale-Invariant Feature Transform (SIFT) algorithm is one of the most widely used image-matching algorithms. It consists of two main parts: Keypoint detection and descriptor extraction. SIFT is considered compute-intensive, leading to the development of many keypoint extraction and description methods that sacrifice SIFT's wide applicability in favor of higher speed. PopSift was implemented using CUDA to perform feature extraction on 1080p images at 25 fps on consumer GPUs such as the NVIDIA GTX 1080, without compromising any details of the SIFT algorithm. It achieves keypoint extraction and description performance that is as accurate as the best existing alternative implementations, such as vlfeat[1]. However, since PopSift was implemented with CUDA, it is vendor-locked to Nvidia hardware and its ecosystem, limiting and excluding its portability to GPUs of other vendors and computers without GPU. While PopSift is an open-source project, vendor lock-in restricts its adoption, knowledge of this field, and use for developers and researchers.

PopSift-SYCL solves this by keeping PopSift's goal of being a faithful open-source SIFT implementation while adding cross-vendor compatibility. It can be ported to a broader range of devices supported by either the Intel DPC++ compiler or the open-source AdaptiveCPP compiler, thereby making it available for systems with AMD, Intel, or Nvidia GPUs. In this paper, we present the stages of the SIFT algorithm, which present a variety of parallelization opportunities, explain the PopSift-SYCL port, and compare its performance on several platforms with its CUDA ancestor.

2 The SIFT Algorithm

SIFT is a multi-stage feature extraction algorithm for detecting and describing local image features in a scale- and rotation-invariant manner, appreciated for its ability to match also across moderate 3D rotation of the scene. This section presents a high-level overview of the SIFT pipeline, focusing on the core algorithmic steps. Pseudocode is used selectively to illustrate the algorithm's logical structure. Mathematical and implementation-specific details are intentionally omitted.

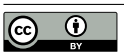
The SIFT algorithm consists of the following main stages:

- Upscaling
- Creating a Gaussian pyramid
- Computing the Difference-of-Gaussian (DoG)
- Detecting keypoints
- Computing the keypoint orientations
- Extracting descriptors
- Normalizing descriptors

Upscaling. Upscale the input image by a factor of 2 in both X and Y dimensions. It is needed to detect features at the highest level of detail, but is often omitted to gain speed.

Creating a Gaussian pyramid. SIFT achieves approximate scale invariance by constructing a Gaussian pyramid of images at multiple scales as illustrated in algo. 1. The pyramid consists of several octaves, where each octave contains a set of images at the same resolution but with progressively increasing Gaussian blur; these images are referred to as levels. The first octave has the resolution of the upscaled input image, and each subsequent

[1] https://vlfeat.org



IWOCL '26, Heilbronn, Germany

ACM ISBN 979-8-4007-2499-2/2026/05
https://doi.org/10.1145/3811257.3811261

Algorithm 1: Build Gaussian Pyramid

```
Input: input_image, num_octaves, num_levels
Output: Gaussian pyramid
pyramid ← array[num_octaves][num_levels];
for octave ← 0 to num_octaves-1 do
    // Initialize first level of octave
    if octave = 0 then
        pyramid[0][0] ← GAUSSIAN_BLUR(input_image);
    else
        source ← pyramid[octave-1][num_levels-3];
        pyramid[octave][0] ← DOWNSAMPLE_BY_2(source);
    end
    // Build progressively blurred levels
    // within an octave
    for level ← 1 to num_levels-1 do
        previous ← pyramid[octave][level-1];
        pyramid[octave][level] ←
         GAUSSIAN_BLUR(previous);
    end
end
return pyramid;
```

Algorithm 2: Build Difference-of-Gaussian Pyramid

```
Input: gauss_pyramid, num_octaves, num_levels
Output: dog_pyramid
dog_pyramid ← array[num_octaves][num_levels - 1];
for octave ← 0 to num_octaves-1 do
    for level ← 0 to num_levels-2 do
        g1 ← gauss_pyramid[octave][level];
        g2 ← gauss_pyramid[octave][level+1];
        dog_pyramid[octave][level] ← g2 − g1;
    end
end
return dog_pyramid;
```

Algorithm 3: SIFT Keypoint Detection

```
Input: Difference-of-Gaussian (DoG) pyramid
Output: Detected keypoints
foreach octave in DoG pyramid do
    for z ← 1 to levels − 2 do
        foreach pixel (x, y) in level z do
            // Find local extrema in space
            // and scale
            if DoG(x, y, z) is a maximum or minimum in its
             3×3×3 neighborhood then
                // Refine location in
                // continuous space
                Estimate subpixel offset (Δx, Δy, Δz);
                if offset is too large then
                    continue;
                end
                // Reject low-contrast points
                Compute contrast at refined location;
                if |contrast| < threshold_contrast then
                    continue;
                end
                // Reject edge-like responses
                Estimate edge response from local curvature;
                if edge response exceeds threshold then
                    continue;
                end
                // Accept stable keypoint
                Accept keypoint at refined position;
            end
        end
    end
end
```

octave has half the resolution of the previous one. New octaves are generated by downsampling the third-to-last level of the preceding octave. Gaussian blurring is applied using a 2D Gaussian filter, usually called σ. Consequently, the image that initializes the next octave is downsampled by a factor of two and has an effective blur of 2σ w.r.t. the first level of the previous octave.

Computing the Difference-of-Gaussian (DoG). The DoG pyramid is derived from the Gaussian pyramid by computing the difference between successive Gaussian-blurred images within each octave. Specifically, each DoG level is obtained by subtracting the image at level l from the image at level l+1 as shown in algo. 2. This operation highlights structures that persist across scales and provides an efficient approximation of the scale-normalized Laplacian-of-Gaussian [8], which is used to detect stable keypoints in scale space.

Detecting keypoints. Every pixel in the DoG layers that is an extremum (absolute minimum or maximum) in its neighbourhood across the current, previous, and next scale levels initiates a keypoint search in scale-space. The candidate location is refined to subpixel accuracy by fitting a 3D quadratic function. Candidates are retained only if they exhibit sufficient contrast and are not dominated by edge-like responses, as represented by algo. 3.

Computing the keypoint orientations. Each detected keypoint may be associated with one or more SIFT descriptors. For each descriptor, a dominant orientation is assigned based on the distribution of local image gradients in a neighborhood around the keypoint. Gradient magnitudes and orientations are computed at each pixel in this region, and each gradient contributes to a 36-bin orientation histogram. Contributions are weighted by both the gradient magnitude and their distance from the keypoint center, giving greater weight to gradients near the keypoint.

After populating the histogram, it is smoothed using multiple passes of averaging to reduce noise and create more stable peaks. Dominant orientations are then identified by finding histogram peaks. The global maximum peak is always selected, and up to 3 additional peaks with an accumulated weight of at least 80% of the maximum are selected as secondary orientations. Each selected

Algorithm 4: SIFT Orientation Assignment

Input: Detected keypoints, Gaussian pyramid
Output: Keypoints with assigned orientations

foreach *keypoint at* (x, y, σ) **do**
  Initialize orientation histogram (36 bins);
  Define neighborhood radius proportional to σ;
  foreach *pixel in neighborhood* **do**
    Compute gradient magnitude and orientation;
    Weight contribution by gradient magnitude and distance to keypoint;
    Add weighted contribution to corresponding histogram bin;
  end
  `// Smooth histogram to reduce noise`
  Apply multiple passes of smoothing;
  `// Select dominant orientations`
  Find maximum histogram value;
  foreach *bin with value* $\geq 0.8\times$ *maximum* **do**
    Refine peak position using quadratic interpolation;
    Convert refined bin position to orientation angle;
    Create keypoint descriptor with this orientation;
  end
end

peak is refined to sub-bin accuracy using quadratic interpolation of the peak and its neighboring bins. The refined location of each peak in the histogram is interpreted as an orientation angle, which is used to achieve rotation invariance in the following stage. If multiple orientations are found, the following stage creates one keypoint descriptor for each of them, as illustrated by algo. 4.

Extracting descriptors. For each keypoint with its assigned dominant orientation, a 128-dimensional descriptor is computed to capture the local image structure in a rotation-invariant manner. The descriptor consists of a 4×4 spatial grid of gradient orientation histograms, each with 8 orientation bins, yielding $4 \times 4 \times 8 = 128$ dimensions.

As represented by algo. 5, the descriptor is computed by sampling gradients in a neighborhood around the keypoint, with the sampling region scaled proportionally to the keypoint's scale. PopSift and PopSift-SYCL extract the sample points from the octave and level of the Gaussian pyramid that is the closest to this scale, which may, after refinement, be different from the level in which it was detected. All sampled gradients are rotated relative to the keypoint's dominant orientation to achieve rotation invariance. Each gradient's contribution is weighted with the gradient magnitude, its distance from the keypoint center, and its position relative to the descriptor grid boundaries.

Gradient contributions are smoothly distributed across neighboring bins to ensure robustness to small geometric distortions. Spatially, contributions are distributed across the 4×4 grid using bilinear interpolation based on the gradient's position. In the orientation dimension, contributions are distributed between adjacent orientation bins proportionally to the gradient's direction. This smooth interpolation prevents abrupt changes in the descriptor when gradients fall near bin boundaries.

Normalizing descriptors. After all gradient contributions are accumulated, the resulting 128-dimensional vectors are normalized to unit length, ensuring invariance to illumination changes.

Algorithm 5: SIFT Descriptor Extraction

Input: Keypoints with orientations, Gaussian pyramid
Output: 128-dimensional descriptors

foreach *keypoint at* (x, y, σ, θ) **do**
  `// Initialize descriptor`
  Initialize descriptor array (4×4 spatial grid $\times$ 8 orientation bins);
  Define sampling region scaled by σ;
  `// Sample gradients in neighborhood`
  foreach *pixel in sampling region* **do**
    Compute gradient magnitude and orientation;
    Rotate gradient orientation by $-\theta$ to align with keypoint;
    `// Compute weights`
    Weight by gradient magnitude;
    Weight by distance to keypoint center (Gaussian weighting);
    Weight by position relative to spatial grid (bilinear interpolation);
    `// Distribute contribution to histogram bins`
    Determine spatial grid cell(s) based on pixel position;
    Determine orientation bin(s) based on rotated gradient direction;
    Distribute weighted gradient to neighboring bins using interpolation;
  end
  `// Normalize descriptor`
  Normalize descriptor vector to unit length;
  Store descriptor;
end

3 PopSift implementation

PopSift follows the SIFT pipeline defined by Lowe [9] and is designed as a drop-in replacement for VLFeat [10]. It implements a non-blocking dataflow using an input queue and double-buffered GPU transfers, but offers an API based on C++ `std::future` for applications, allowing it to serve as a drop-in replacement for synchronous CPU libraries.

SIFT extraction is performed asynchronously by two background threads. The first transfer moves jobs from an unbounded queue into a double buffer allocated in CUDA-pinned host memory, enabling efficient DMA transfers while limiting pinned memory usage. The second thread launches the required CUDA kernels to perform feature extraction.

Since the number of extracted features varies significantly, output buffers are allocated dynamically using page-aligned memory.

During download, the buffer is temporarily pinned for DMA, then unpinned and stored in a `std::promise`, potentially unblocking the waiting `future`.

The main stages of the SIFT feature extraction pipeline are illustrated in Figure 1.

Image upload and upscale. Once the input image is transferred to the GPU, it is accessed via a CUDA texture, enabling hardware-supported normalized addressing and bilinear interpolation. This allows processing at arbitrary up- and downscaling factors, rather than being limited to the original or double-resolution image sizes.

Gaussian blurring and DoG computation. PopSift exploits the separability of Gaussian filters, implementing horizontal and vertical convolution passes as separate kernel invocations. Filter coefficients are stored in CUDA constant memory for fast, thread-wide caching. Filter symmetry is leveraged to reduce the number of multiplications by summing pixels with symmetric offset from the center before multiplying them with their common Gaussian weight.

Since multiplication operations are the main bottleneck in Gaussian filter computation, PopSift optionally employs hardware interpolation to further reduce memory accesses and arithmetic operations. The implementation supports multiple methods for computing filter width to balance accuracy and performance.

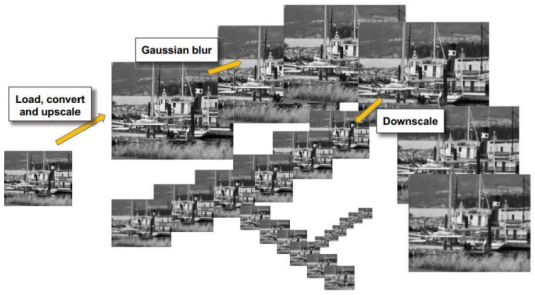
The first octave applies a Gaussian blur while simultaneously resampling from the input image using CUDA's native texture-sampling hardware for efficient bilinear interpolation and sub-pixel sampling. Downscaling between octaves defaults to the prescribed SIFT approach by selecting the third-to-last blur level of the previous octave and subsampling every second pixel. Alternative downscaling modes are available to increase parallelism: downscaling the first level of each octave directly from the upscaled input image, or downscaling all levels of an octave directly from the scaled input image.

After Gaussian blurring completes, DoG layers are computed by subtracting consecutive blur levels pixel-wise.

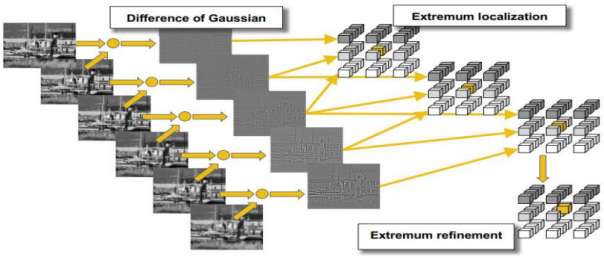
Keypoint detection. PopSift employs a straightforward thread-mapping scheme in which each CUDA thread checks a single pixel for extrema. Threads are arranged in blocks of 32×4, where the width of 32 achieves optimal memory coalescing through 128-byte aligned loads, and the height of 4 provides sufficient parallelism to hide memory access latency.

Extremum computation evaluates whether a pixel is either an absolute minimum or maximum among its 26 neighbors in the 3D scale-space (8 neighbors in the same blur level, plus 9 neighbors each in the levels above and below). This evaluation uses bitmask operations to avoid conditional branches, enabling 32-thread warps to execute in lockstep and maximizing GPU occupancy. The presence of an extremum is determined through a single Boolean operation on the computed bitmask.

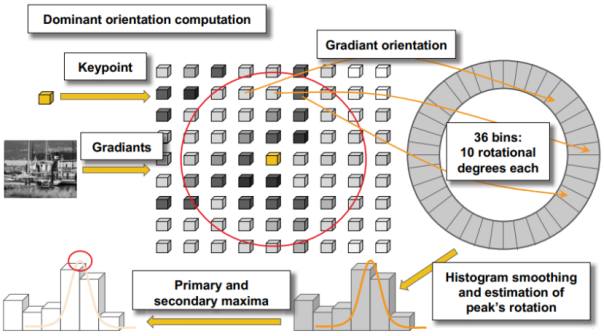
Subpixel refinement of keypoint positions is performed in three dimensions: spatial X and Y coordinates, and scale (blur level). PopSift employs a closed-form analytical solution rather than iterative Gaussian elimination, thereby improving computational efficiency. The implementation provides multiple refinement variants with different behaviors.



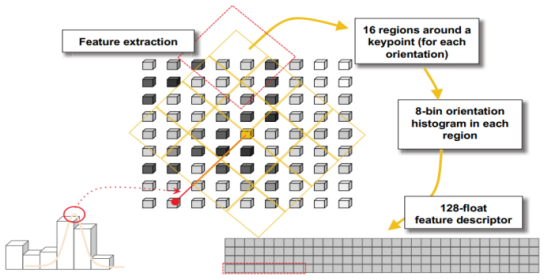
(a) Gaussian pyramid construction



(b) DoG computation and extrema localization



(c) Orientation assignment



(d) Descriptor construction

Figure 1: Main stages of the SIFT feature extraction pipeline. Reproduced from [5].

Dominant orientation computation. PopSift leverages CUDA warp-level primitives to optimize the computation of the dominant orientation. The implementation employs fast intrinsic operations including shuffle instructions (`__shfl*()`), population count

(`__popc()`), and warp voting (`__ballot()`) for efficient intra-warp communication and reduction operations. This approach constrains the number of threads to powers of two, with a maximum of 32 threads per warp.

The standard SIFT algorithm uses 36 histogram bins to collect an orientation histogram of gradients around each extremum, each representing 10°. PopSift uses 16 threads per keypoint rather than allocating one thread per bin, trading reduced thread count for lower memory latency and better utilization of warp-level operations. Each thread accumulates contributions from multiple histogram bins, utilizing warp-shuffle operations for efficient inter-thread data exchange within the warp.

To efficiently identify secondary dominant orientations, PopSift implements a specialized 32-element bitonic sort. This fixed-size sort operates entirely within warp-level registers, avoiding global memory accesses and enabling rapid identification of the strongest orientation peaks that exceed 80% of the maximum histogram value, as prescribed by the SIFT algorithm.

Descriptor extraction. PopSift implements three approaches for descriptor computation.

In the "loop" approach, 512 threads are organized into 16 groups of 32, where each group computes an 8-bin histogram for one of the 16 rotated squares surrounding a keypoint. Each group defines an axis-aligned bounding box containing its assigned square and scans every pixel, accumulating weighted gradient information for pixels inside the square into the histogram.

The "grid" approach employs texture sampling to sample each of the 16 squares following a 16×16 grid pattern aligned to the keypoint's dominant orientation. Groups of 256 threads cooperate per keypoint orientation, organized into 16 groups per grid square, computing weighted gradients and inserting them into histograms.

The "notile" approach exploits overlap between squares. While corner-square regions contribute only to their own histogram, pixels near the center may contribute to up to 4 histograms. This method uses 32 threads per keypoint to sample a 64×64 grid covering all 16 squares, computing each gradient once and distributing it to the relevant histograms based on spatial proximity.

All approaches conclude with descriptor normalization. PopSift implements both L2 and RootSIFT normalization variants. Descriptors can optionally be scaled by powers of two during normalization. This enables an efficient conversion from floating-point to byte representation of the final descriptors before transfer to the host. Typically, users choose scale factors 2^8 or 2^9.

4 PopSift-SYCL Implementation

PopSift-SYCL is a port designed to compare SYCL's portability model with CUDA while achieving Single Source Multiple Compilers (SSMC) compatibility. The core algorithmic approach follows Lowe [9] identically to the CUDA version [5], but the implementation makes deliberate simplifications to focus on kernel performance comparison rather than reproducing all CUDA-specific optimizations.

Architectural Simplifications. For research clarity, the SYCL version adopts a synchronous processing model. The two-threaded asynchronous dataflow, double buffering, and pinned memory optimizations from the CUDA version are intentionally omitted. Jobs move through two queue stages that maintain API compatibility with `futures`. These simplifications isolate SYCL kernel performance from host-side pipeline engineering, enabling direct comparison of device execution characteristics.

Image Handling and Memory. PopSift-SYCL does not utilize `sycl::image` for texture operations, as AdaptiveCPP, one of the target compilers, lacks support for this feature. Consequently, hardware-accelerated bilinear filtering used in Gaussian pyramid creation, normalized coordinates used in image up- and downscaling, and texture-optimized reads are unavailable. Instead, the implementation uses standard SYCL buffers with direct memory access. Backend-specific optimizations (e.g., CUDA texture interoperability provided by DCP++'s `sycl::image`) are possible but would compromise the SSMC objective. Memory transfers use standard SYCL buffer operations without pinned memory or DMA optimizations.

Research Trade-offs. The implementation prioritizes comparing SYCL and CUDA computation models across multiple backends (AdaptiveCPP, DPC++) over maximizing absolute performance. A production-grade SSMC implementation supporting all hardware features across backends would require substantial engineering effort with backend-specific fallbacks and interoperability layers—beyond the scope of this comparative study. The resulting codebase demonstrates SYCL's portability while quantifying performance differences attributable to the programming model itself rather than auxiliary optimizations.

Gaussian blurring and DoG computation. PopSift-SYCL exploits the separability property of the Gaussian filter, implementing horizontal and vertical convolution passes as separate kernel invocations. Like the original CUDA implementation, filter symmetry is leveraged to reduce the number of multiplications by accumulating symmetric offset pairs with shared weights before adding the center tap.

The first octave applies a Gaussian blur while simultaneously resampling from the input image. Since SYCL does not provide native texture sampling hardware as CUDA does, bilinear interpolation is implemented explicitly in device code to perform sub-pixel sampling and avoid aliasing artifacts during upscaling. For each output pixel, source coordinates are computed via precomputed scale factors, and the symmetric horizontal filter is applied using these bilinearly-interpolated samples.

Downscaling between octaves follows the prescribed SIFT approach by selecting the third-to-last blur level of the previous octave and subsampling every second pixel. This maintains computational efficiency while adhering to the standard scale-space construction.

After Gaussian blurring completes, DoG layers are computed via pixel-wise subtraction between consecutive blur levels. The implementation follows a two-phase sequential pipeline: all Gaussian filtering across all octaves completes before any DoG computation begins. This design mirrors the original CUDA implementation to enable direct performance comparison between the two stages. But unlike the original CUDA implementation, which launches DoG kernels sequentially across octaves using separate CUDA streams and synchronizes after each octave, PopSift-SYCL achieves higher parallelism by leveraging a unified execution model. All DoG kernels across all octaves are submitted to a single shared SYCL queue

in rapid succession. This design allows the SYCL runtime to automatically schedule and overlap kernel execution across available GPU resources without explicit stream management.

The shared queue approach offers several advantages over multiple streams: it eliminates the overhead of stream creation and management, simplifies dependency tracking through SYCL's event-based model, and allows the runtime to dynamically optimize resource allocation based on current GPU utilization. Kernel completion is tracked through lightweight SYCL events that are collected in a vector and synchronized only once at the end of all submissions, rather than synchronizing after each octave. This asynchronous batch submission followed by bulk synchronization maximizes GPU occupancy and reduces host-device synchronization overhead, resulting in improved overall performance compared to the sequential stream-based approach.

Keypoint detection. PopSift-SYCL maintains the one-thread-per-pixel approach for keypoint detection. Work-items are launched in a three-dimensional grid corresponding to the DoG pyramid structure: width × height × (levels−3), with each work-item examining a single spatial position at one blur level. The implementation preserves the bitmask-based extremum detection from the CUDA version, evaluating whether a pixel is an absolute minimum or maximum among its 26 scale-space neighbors through progressive bit operations.

Before extremum testing, an initial contrast threshold filter eliminates low-contrast candidates. Subpixel refinement then iteratively adjusts the keypoint position in X, Y, and scale dimensions. The refinement process allows keypoints to migrate between blur levels, consistent with the CUDA implementation's behavior. Refinement terminates after a maximum number of iterations or when the computed offset falls below the migration threshold in all dimensions.

Final validation applies edge response filtering. Extrema that pass all tests are written to device memory using atomic operations to maintain a per-octave count. All octaves execute their extrema detection kernels asynchronously, with synchronization deferred until all octaves complete. Results are then copied to host memory and consolidated into the pyramid's extrema structure.

Dominant orientation computation. PopSift-SYCL adopts a one-work-item-per-extremum approach for dominant orientation computation. Each work-item constructs a 36-bin orientation histogram by examining gradients in a circular region around its assigned extremum. Gradient magnitudes are weighted by a Gaussian function with σ_w and accumulated into histogram bins representing 10° intervals.

Histogram smoothing uses weighted averaging across neighboring bins. Peak locations are refined by quadratic fitting around local maxima, and additional orientations are assigned when secondary peaks exceed 80% of the maximum histogram value.

All octaves execute their orientation kernels asynchronously on a shared SYCL queue. Results are accumulated into a single device buffer using atomic operations, with synchronization deferred until all octaves complete. A prefix sum computation on the host establishes the mapping between extrema and their multiple orientations.

Descriptor extraction. PopSift-SYCL implements the VLFeat-style "loop" approach where each keypoint orientation is processed by a single work-item. For each pixel, the work-item computes gradients using finite differences and performs trilinear interpolation to distribute Gaussian-weighted gradient contributions across the $4 \times 4 \times 8$ descriptor histogram, with gradients rotated by the keypoint's dominant orientation for rotation invariance.

The implementation uses unified memory, allocating all descriptor memory once before processing any octave. Shared data structures (extrema and feature-to-extremum mappings) transfer to the device once and are reused across octaves via event-based dependencies. All octave extraction kernels launch asynchronously on a shared SYCL queue, writing directly to their respective regions in the unified descriptor buffer using pointer offsets, eliminating intermediate buffers and per-octave allocation overhead.

Following extraction, descriptor normalization applies either L2 or RootSIFT in a single operation. Both scale descriptors by powers of two (typically 2^8 or 2^9) to enable floating-point to byte conversion. The entire normalized descriptor set transfers to the host in one memory copy, minimizing overhead and improving throughput.

5 Compiling and Using PopSift-SYCL

PopSift-SYCL was developed and tested on the Simula EX3 HPC cluster using two SYCL implementations: Intel oneAPI DPC++ (2025.1.0) for Intel GPU Max 1100, and AdaptiveCpp (25.10.0) for AMD Instinct MI210 and NVIDIA Tesla V100. An additional validation run on NVIDIA RTX 4050 (Ubuntu 24.04.2) was also performed with AdaptiveCpp (25.10.0). The implementation has five mandatory dependencies: a SYCL 2020-compliant compiler as the GPU programming framework, CMake 3.22+ for the build system, a C++17-compatible toolchain, Boost 1.73.0+, and a vendor-specific runtime (Level Zero for Intel, ROCm for AMD, or CUDA for NVIDIA). OpenImageIO support has been disabled due to SYCL incompatibility; only PGM and PPM image formats are supported.

To compile PopSift-SYCL, clone the repository from https://github.com/alicevision/popsift/tree/sycl. The build system automatically detects the SYCL compiler implementation and target GPU architecture using `nvidia-smi` for NVIDIA GPUs, `rocminfo` for AMD GPUs, or runtime selection for Intel devices. Set up the build using CMake:

```
cmake -S . -B build -DCMAKE_CXX_COMPILER=<compiler>
cmake --build build -j
```

where `<compiler>` is `icpx` for Intel DPC++, `acpp` for AdaptiveCPP.

The main outputs are the library `libpopsift.so`, headers, and the demo program `popsift-demo`. The only mandatory parameter is the input image, provided via the `-i` option, which takes the image path as its argument (e.g., `./popsift-demo -i <testImage>`).

Compilation flows used in this work. To improve reproducibility, we report compilation flow in addition to compiler version. For AdaptiveCpp on NVIDIA/AMD, the benchmark binaries were built in explicit target mode (CUDA/HIP target-specific flow), and not in generic SSCP flow. This is reflected by the runtime/compiler configuration (`plugin-with-sscp-compiler=false`) in the AdaptiveCpp installations used for experiments. For Intel, DPC++ was used with `-fsycl` and Level Zero runtime selection.

Why SSCP results are not included. We attempted to build an SSCP-enabled AdaptiveCpp toolchain on the cluster, but could not establish a stable, reproducible build environment during the benchmark campaign due to missing/incompatible system toolchain dependencies in scheduled jobs. Therefore, SSCP/generic-flow timings are not reported in this paper.

Vendor-compiler coverage. Not all vendor–compiler combinations could be included in the benchmark matrix. In particular, some combinations were not supported by the available software stack on the cluster (or could not be built in a stable, reproducible way within the campaign window), similar to the SSCP case above. Therefore, we report only combinations that were both functional and reproducible under identical job-scheduler conditions. The selected set still covers three major GPU vendors and two SYCL implementations, providing a representative cross-platform comparison.

Table 1 summarizes the evaluated configurations.

Table 1: SYCL compiler, backend, and compilation-flow summary.

Vendor	Compiler	Backend	Flow
Intel	DPC++	Level Zero	`-fsycl` runtime-selected
AMD	AdaptiveCpp	ROCm (HIP)	Explicit target (`hip:gfx90a`)
NVIDIA	AdaptiveCpp	CUDA	Explicit target (`cuda:sm_70`)

These configurations highlight the portability offered by the SYCL programming model. PopSift-SYCL is compiled from a single source code base across all evaluated platforms, with differences limited to compiler selection and backend-specific dependencies. This separation allows performance and scalability to be evaluated independently of algorithmic changes, which is essential for systematic benchmarking on heterogeneous architectures.

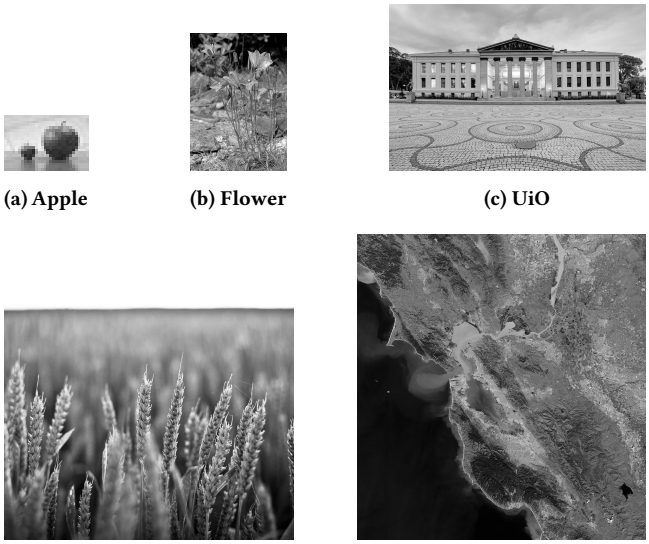
6 Results: Benchmarks and Profiling

To evaluate the performance and portability of PopSift-SYCL, we conducted extensive benchmarks across heterogeneous hardware platforms using two SYCL compiler toolchains . Experiments were performed on a PC with an Nvidia RTX 4050 GPU and on an HPC cluster with different GPUs. Our test configurations consisted of:

- **AdaptiveCpp + NVIDIA V100**: Data center GPU
- **AdaptiveCpp + NVIDIA RTX 4050**: Desktop GPU
- **AdaptiveCpp + AMD MI210**: Data center GPU
- **DPC++ + Intel Max 1100**: Data center GPU

Testing used five images with varying resolutions to assess both constant-time (pyramid construction) and feature-dependent (descriptor extraction) components, are shown in Fig. 2:

- `apple.pgm`: 30×20 pixels (6 features) — minimal workload
- `blomst.pgm`: 360×515 pixels (~2,200 features) — low resolution
- `uio.pgm`: 1250×812 pixels (~7,200 features) — medium resolution
- `Wheat.pgm`: 1939×2048 pixels (~7,700 features) — high resolution
- `SF.pgm`: 1939×2048 pixels (~21,000 features) — high resolution & feature-rich



(a) Apple **(b) Flower** **(c) UiO**

(d) Wheat **(e) SF**

Figure 2: Test images shown with relative size emphasis (not to exact scale)

Image	Feature Points		Descriptors	
	Absolute	Relative	Absolute	Relative
apple.pgm	0	0.00%	0	0.00%
blomst.pgm	−3	−0.13%	−33	−1.23%
uio.pgm	+1	+0.01%	−31	−0.36%
Wheat.pgm	−6	−0.08%	−53	−0.60%
SF.pgm	−30	−0.14%	−398	−1.49%

Table 2: Feature detection differences: SYCL − CUDA. Negative values indicate fewer features in SYCL.

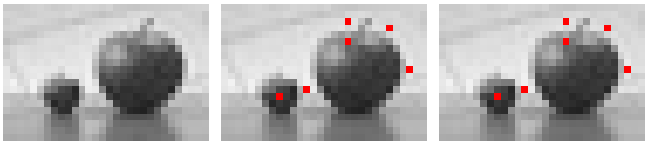
6.1 Correctness Validation

We use the original CUDA implementation as the correctness baseline, comparing the SYCL port using two criteria: (1) feature and descriptor counts, and (2) feature properties (coordinates, σ values, descriptor vectors). Both implementations produce deterministic outputs.

6.1.1 Feature Count Analysis. Table 2 summarizes feature detection differences between CUDA and SYCL. The SYCL implementation produces identical results across all four platforms (NVIDIA RTX 4050, NVIDIA V100, AMD MI210, Intel Max 1100), demonstrating deterministic, portable behavior across three GPU vendors.

Feature point differences range from −0.14% to +0.01%, while descriptor differences range from −1.49% to 0%. These stem from variations in floating-point precision and threshold comparisons near decision boundaries. The largest discrepancy (`SF.pgm`: 30 fewer features, 398 fewer descriptors) represents < 1.5% difference – this is within acceptable bounds for SIFT applications.

6.1.2 Descriptor-Level Validation. Examining `apple.pgm`'s six features reveals that all setups detect features at identical spatial locations. An example of it is shown in Fig. 3.



(a) Original (b) CUDA (RTX4050) (c) SYCL (RTX 4050)

Figure 3: Feature detection comparison on the apple image

All SYCL setups (NVIDIA V100/RTX 4050, Intel Max 1100, AMD MI210) match coordinates to sub-pixel precision and produce σ values within 0.002% of CUDA. Descriptor component variations range from 10^{-4} to 10^{-3}, with NVIDIA SYCL backends showing minimal deviation from the CUDA baseline, while Intel and AMD exhibit slightly larger but equivalent variations due to different math library implementations. All SYCL setups produce virtually identical results to each other despite different hardware architectures and compilers.

6.2 CUDA vs. SYCL on NVIDIA Hardware Performance Analysis

Figures 4 and 5 present a direct comparison between our SYCL implementation and the original CUDA PopSift on the RTX 4050 GPU. The results indicate a substantial performance gap, with CUDA consistently outperforming SYCL by factors ranging from 1.9× to 6.9× across all test images. On the UIO image, CUDA completed the pipeline in 26.6 ms compared to 64.6 ms for SYCL (2.4× slowdown), while the SF image showed execution times of 82.4 ms versus 156.2 ms, respectively (1.9× slowdown). The smallest image (APPLE) exhibited the highest relative slowdown at 6.9× (0.6 ms vs 4.3 ms).

The stage-level breakdown reveals significant non-uniformity in performance overhead across the pipeline. The Gaussian Filter stage demonstrates the most pronounced disparity, with slowdown factors ranging from 6.6× on APPLE to 95.05× on WHEAT. Similarly, the Find Extrema stage exhibits slowdowns between 6.1× and 97.8×. In contrast, the Normalization and the DoG stage actually perform faster in SYCL (0.01×–1.4×) and (0.2×–2.0×). The Descriptor Extraction and Orientation stages demonstrate moderate overhead at 4.5×–20.7× and 1.7×–4.5× respectively.

A potential avenue for performance improvement lies in implementing `sycl::image` support in AdaptiveCpp, which would enable utilization of hardware-accelerated texture operations. The CUDA implementation leverages CUDA texture memory for bilinear filtering, normalized coordinate access, and cache-optimized memory reads—particularly critical for the Gaussian Filter and Find Extrema stages that perform intensive interpolated sampling. Our SYCL implementation relies on standard buffer access with manual interpolation, resulting in higher memory latency and reduced cache efficiency. Additionally, differences in kernel launch overhead, work-group scheduling strategies, and compiler optimization between nvcc and AdaptiveCpp's SYCL-to-CUDA translation layer contribute to the observed performance gap. The non-uniform overhead across stages suggests that memory-bound operations with irregular access patterns suffer disproportionately under the current SYCL implementation constraints.

6.3 Benchmarking on the different devices Performance Analysis

Figure 6 presents a comprehensive comparison of the SIFT pipeline execution times across multiple hardware platforms and implementations. Our SYCL port was evaluated on AMD MI210, NVIDIA V100, and NVIDIA RTX 4050 GPUs via AdaptiveCpp (ACPP), as well as an Intel Max 1100 GPU using DPC++. The results reveal significant performance variations both across different hardware architectures and between implementation approaches.

On AMD's MI210 GPU, execution times ranged from 14 ms for the simplest image (APPLE) to 148 ms for the most complex (SF), with the Gaussian Filter and Descriptor Extract stages consuming the majority of execution time. In contrast, NVIDIA's V100 achieved significantly better performance, completing the same workloads in 8–116 ms, demonstrating superior computational throughput particularly in the memory-intensive Gaussian filtering stage. The RTX 4050, despite being a consumer-grade GPU, showed competitive performance with execution times between 4–156 ms.

The Intel Max 1100 running DPC++ exhibited the highest execution times (55–373 ms), which can be attributed to several factors: architectural differences in the Xe-HPC compute architecture, potential suboptimal kernel tuning for Intel's platform, and the relative maturity of the DPC++ compiler toolchain compared to CUDA. Notably, the performance scaling pattern across images remains consistent across all platforms, with the SF image consistently requiring the longest execution time due to its higher feature count and larger dimensions.

7 Conclusion, Discussion and Future Work

7.1 Conclusion

This work successfully demonstrates SYCL's portability promise by implementing PopSift, a complete, faithful SIFT algorithm, across AMD, Intel, and NVIDIA GPUs using two independent SYCL compilers (AdaptiveCpp and DPC++). The implementation achieves Single Source Multiple Compilers (SSMC) compatibility, confirming that SYCL can deliver on its cross-platform vision for real-world computer vision workloads.

Correctness Validation. Our SYCL implementation produces deterministic, functionally equivalent results across all four tested platforms (AMD MI210, Intel Max 1100, NVIDIA V100, NVIDIA RTX 4050). Feature detection differences remain relativly low for keypoints and for descriptors when compared to the CUDA baseline, with all SYCL backends producing virtually identical outputs regardless of hardware vendor or compiler. This validates SYCL's abstraction model for numerical consistency across heterogeneous architectures.

Performance Analysis. Direct comparison on identical NVIDIA hardware (RTX 4050) reveals substantial performance gaps: CUDA outperforms SYCL by 1.9–6.9× across test images. However, this overhead is non-uniform and attributable to specific implementation constraints rather than fundamental SYCL limitations. A primary bottleneck stems from AdaptiveCpp's lack of `sycl::image` support, preventing hardware-accelerated texture memory operations critical for interpolation-heavy stages. Gaussian Filter and Find Extrema stages exhibit 6–98× slowdowns due to manual buffer

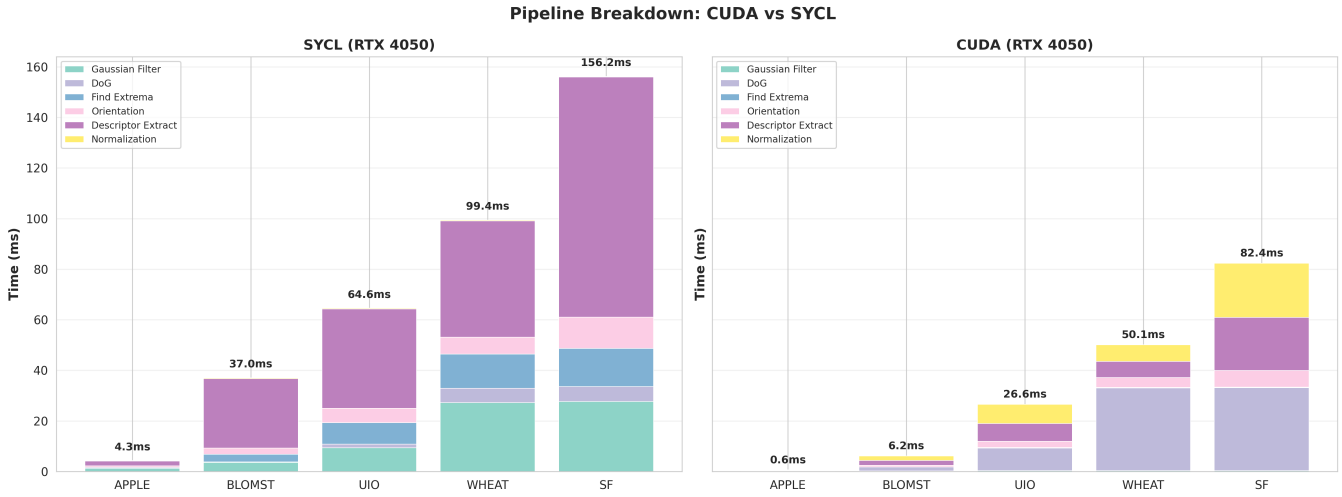


Figure 4: Pipeline breakdown comparison between CUDA and SYCL implementations on NVIDIA RTX 4050. Each stacked bar shows the execution time decomposed into pipeline stages (Gaussian Filter, DoG, Find Extrema, Orientation, Descriptor Extract, and Normalization) for five test images of varying sizes. Total execution times are displayed above each bar, with CUDA consistently outperforming SYCL across all images.

SYCL Slowdown Factor (SYCL time / CUDA time)

Stage	APPLE	BLOMST	UIO	WHEAT	SF
Gaussian Filter	6.62×	14.02×	29.75×	95.05×	89.28×
DoG	1.99×	0.20×	0.16×	0.18×	0.18×
Find Extrema	6.09×	41.20×	45.93×	92.63×	97.79×
Orientation	4.49×	3.90×	2.25×	1.72×	1.89×
Descriptor Extract	20.69×	14.89×	5.66×	7.24×	4.50×
Normalization	1.39×	0.17×	0.04×	0.06×	0.01×
Total Pipeline	**6.87×**	**6.00×**	**2.43×**	**1.98×**	**1.90×**

Color code: Green (≤1.0×) | Yellow (1.0-2.5×) | Red (>2.5×)

Figure 5: Per-stage slowdown factors (SYCL time / CUDA time) on NVIDIA RTX 4050. Values ≤1.0 (green) indicate SYCL is faster, while values >1.0 indicate CUDA is faster. The table reveals that Gaussian Filter and Find Extrema stages exhibit the largest slowdowns (6–98×) due to lack of texture memory support in SYCL, while DoG and Normalization stages perform competitively or better in SYCL. Overall pipeline slowdown ranges from 1.9× to 6.9×.

interpolation replacing texture hardware, while other stages (DoG, Normalization) achieve near-parity or even outperform CUDA. This suggests that with mature texture memory support, the performance gap could narrow significantly for memory-bound operations.

Research Trade-offs. A production implementation incorporating asynchronous dataflow, backend-specific texture interoperability, and optimized memory transfers would narrow the CUDA performance gap while potentially compromising SSMC purity.

7.2 Discussion

The Portability-Performance Tension. PopSift-SYCL demonstrates that SYCL's portability is achievable but comes with measurable performance costs when compiler ecosystems lack feature completeness. The absence of standardized texture memory abstractions in current SYCL specifications forces implementations to choose between (1) vendor-specific interoperability paths that break SSMC, (2) performance penalties from buffer-based workarounds, or (3) waiting for ecosystem maturation. Our SSMC-first approach quantifies option (2): 2–7× overhead on NVIDIA hardware for a compute-intensive algorithm with heavy texture usage.

Compiler Ecosystem Maturity. The performance hierarchy (CUDA > SYCL/AdaptiveCpp/NVIDIA > SYCL/AdaptiveCpp/AMD > SYCL/DPC++/Intel) reflects both hardware capabilities and compiler optimization maturity. AdaptiveCpp's NVIDIA backend benefits from targeting mature CUDA runtimes, while Intel's DPC++ toolchain shows room for optimization improvements. This gap is expected to narrow as SYCL compilers mature and vendor-specific optimizations improve.

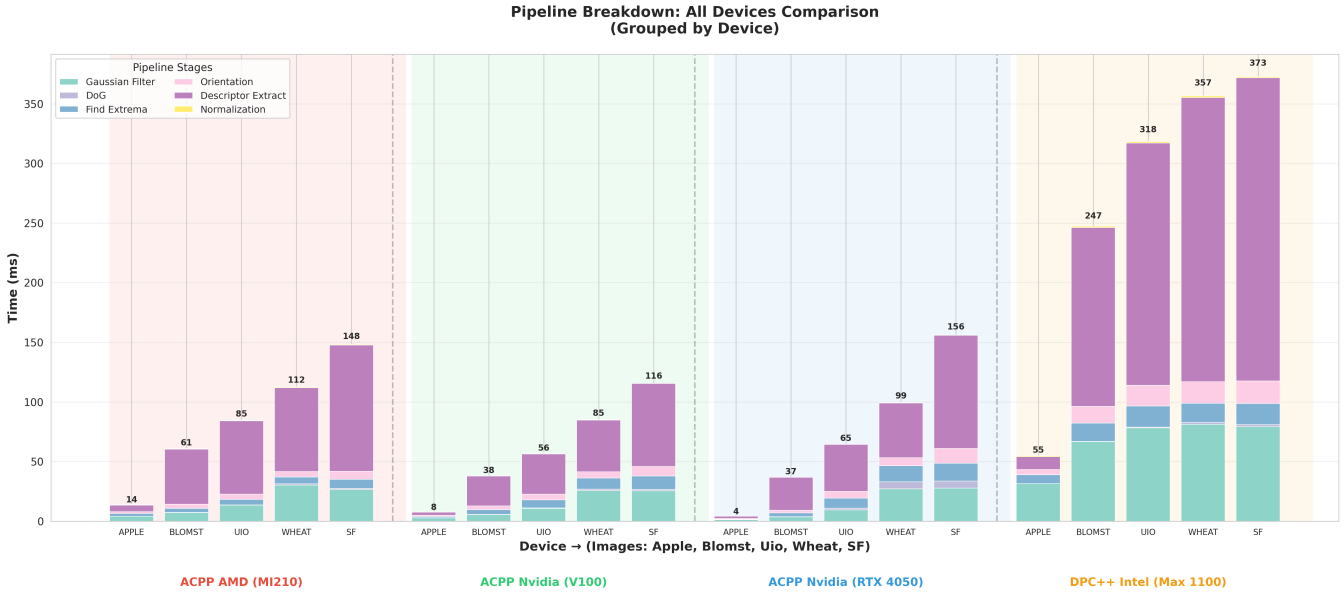


Figure 6: Pipeline breakdown across different GPU architectures and implementations. Results are grouped by device type (AMD MI210, NVIDIA V100, NVIDIA RTX 4050 via ACPP (AdaptiveCpp), and Intel Max 1100 via DPC++) for five test images. Each stacked bar shows the contribution of individual pipeline stages to total execution time. The SYCL implementation shows consistent scaling patterns across all platforms.

Practical Implications. For applications prioritizing broad hardware access over peak performance, PopSift-SYCL's 2–7× slowdown may be acceptable given the 3-vendor portability. For latency-critical deployments on known hardware, native CUDA remains superior.

7.3 Future Work

Texture Memory Support. The most impactful optimization involves implementing `sycl::image` when AdaptiveCpp support becomes available, or developing backend-specific texture interoperability layers with compile-time selection. This could reduce Gaussian Filter and Find Extrema overhead from 6–98× to near-parity based on our stage-wise analysis.

Asynchronous Pipeline. Restoring the two-stage asynchronous dataflow with overlapped CPU-GPU transfers and double buffering would improve throughput without compromising cross-platform compatibility. This represents pure engineering effort rather than fundamental portability barriers.

Extended Benchmarking. Testing on additional platforms would validate SYCL's portability claims beyond discrete GPUs. Performance profiling with vendor tools such as NVIDIA Nsight and Intel VTune, could identify micro-architectural bottlenecks.

Feature Parity Investigation. The 0.14–1.49% descriptor differences warrant investigation of floating-point math library variations across backends. While within acceptable tolerances, understanding these sources could inform algorithm robustness improvements.

Production Hardening. A production-grade PopSift-SYCL would require error handling, multi-device support, batch processing, and performance monitoring, infrastructure omitted in this research prototype.

In conclusion, PopSift-SYCL validates SYCL as a viable path to GPU portability for complex algorithms while illuminating specific ecosystem gaps (texture memory abstractions, compiler maturity) that impact performance. The 2–7× performance cost represents a measurable but potentially acceptable trade-off for applications valuing vendor independence and future-proof hardware access.

References

[1] [n. d.] AdaptiveCpp. GitHub. Retrieved Jan. 11, 2026 from https://github.com/AdaptiveCpp.

[2] AliceVision Project. 2026. Alicevision: photogrammetric computer vision framework. https://alicevision.org/. Accessed: 2026-01-11. (2026).

[3] Aksel Alpay and Vincent Heuveline. 2020. SYCL beyond OpenCL: the architecture, current state and future direction of hipSYCL. In *Proceedings of the International Workshop on OpenCL* (IWOCL '20). Association for Computing Machinery, New York, NY, USA, (Apr. 27, 2020), 1. ISBN: 978-1-4503-7531-3. doi:10.1145/3388333.3388658.

[4] [n. d.] Compile cross-architecture: intel® oneAPI DPC++/c++ compiler. Intel. Retrieved Jan. 11, 2026 from https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html.

[5] Carsten Griwodz, Lilian Calvet, and Pål Halvorsen. 2018. Popsift: a faithful SIFT implementation for real-time applications. In *Proceedings of the 9th ACM Multimedia Systems Conference*. MMSys '18: 9th ACM Multimedia Systems Conference. ACM, Amsterdam Netherlands, (June 12, 2018), 415–420. ISBN: 978-1-4503-5192-8. doi:10.1145/3204949.3208136.

[6] Carsten Griwodz, Simone Gasparini, Lilian Calvet, Pierre Gurdjos, Fabien Castan, Benoit Maujean, Gregoire De Lillo, and Yann Lanthony. 2021. AliceVision meshroom: an open-source 3d reconstruction pipeline. In *Proceedings of the 12th ACM Multimedia Systems Conference* (MMSys '21). Association for Computing Machinery, New York, NY, USA, (Sept. 22, 2021), 241–247. ISBN: 978-1-4503-8434-6. doi:10.1145/3458305.3478443.

[7] Ronan Keryell, Maria Rovatsou, and Lee Howes. [n. d.] Khronos® SYCL™ working group.

[8] Tony Lindeberg. 1994. Scale-space theory: a basic tool for analyzing structures at different scales. *Journal of Applied Statistics*, 21, 1, (Jan. 1994), 225–270. doi:10.1080/757582976.

[9] David G. Lowe. 2004. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60, 2, (Nov. 2004), 91–110. doi:10.1023/B:VISI.0000029664.99615.94.

[10] Andrea Vedaldi and Brian Fulkerson. 2010. Vlfeat: an open and portable library of computer vision algorithms. In *Proceedings of the 18th ACM international conference on Multimedia* (MM '10). Association for Computing Machinery, New York, NY, USA, 1469–1472. ISBN: 978-1-60558-933-6. doi:10.1145/1873951.1874249.