Analyzing Program Flow within a Many-Kernel OpenCL Application

Perhaad Mistry
Electrical and Computer Engg.
Northeastern University
Boston MA
pmistry@ece.neu.edu

Chris Gregg
Computer Science
University of Virginia
Charlottesville VA
chg5w@virginia.edu

Norman Rubin
Advanced Micro Devices
Boxborough, MA
norman.rubin@amd.com

David Kaeli
Electrical and Computer Engg.
Northeastern University
Boston, MA
kaeli@ece.neu.edu

Kim Hazelwood
Computer Science
University of Virginia
Charlottesville, VA

ABSTRACT

Many developers have begun to realize that heterogeneous multi-core and many-core computer systems can provide significant performance opportunities to a range of applications. Typical applications possess multiple components that can be parallelized; developers need to be equipped with proper performance tools to analyze program flow and identify application bottlenecks. In this paper, we analyze and profile the components of the Speeded Up Robust Features (SURF) Computer Vision algorithm written in OpenCL. Our profiling framework is developed using built-in OpenCL API function calls, without the need for an external profiler. We show we can begin to identify performance bottlenecks and performance issues present in individual components on different hardware platforms. We demonstrate that by using run-time profiling using the OpenCL specification, we can provide an application developer with a fine-grained look at performance, and that this information can be used to tailor performance improvements for specific platforms.

Categories and Subject Descriptors

C.1.3 [**Processor Architecures**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Profiling, Performance Measurement

Keywords

OpenCL, Profiling, GPGPU, Heterogeneous computing, Computer Vision, SURF, Performance tools

1. INTRODUCTION

Over the past several years we have seen rapid adoption of general purpose processing moving to new heterogeneous desktops, laptops, and mobile systems that include processors with multiple cores. Stream programming languages such as AMD's Brook [6], NVIDIA's CUDA programming language [1] and, more recently, OpenCL [19] have enabled relatively straightforward development of parallel applications that can run on commodity hardware across a number of platforms. OpenCL is an open standard maintained by the Khronos group, and has received the backing of major graphics hardware vendors.

In heterogeneous computing, knowledge about the architecture of the targeted set of devices is critical to reap the full benefits of the hardware. For example, selected kernels in an application may be able to exploit vectorized operations available on the targeted device, and if some of the kernels can be optimized with vectorization in mind, the overall application may be sped up significantly. However, it is important to gauge the contributions of each kernel to the overall application runtime. Then informed optimizations can be applied to obtain the best performance.

When utilizing a portable programming framework like OpenCL, we also need the ability to assess the benefits of each optimization from a consistent target-neutral interface. Given the fact that OpenCL allows us to choose the specific kernels to execute at runtime, it is important to be able to make choices across a range of target architectures.

Application profiling can be used effectively to guide optimizations. A profiling tool can allow a developer to measure the execution time of the entire application, broken down on an individual kernel basis. In a typical heterogeneous computing scenario, an application starts out executing on the CPU, and then the CPU launches kernels on a second device (e.g., a GPU). The data transferred between these devices must be managed efficiently to minimize the impact of communication. Data manipulated by multiple kernels should be kept on the same device where the kernels are run. In many cases, data is transferred back to the CPU host, or integrated into CPU library functions. Analysis of program flow can pinpoint sections of the application where it would be beneficial to modify data management, leading to more efficient use of the overall system.

1.1 SURF

In this paper, we demonstrate the utility of our profiling framework using the SURF [4]. We were interested in developing a very efficient parallelized version of the SURF framework in OpenCL. SURF represents a large class of applications where profiling and profile-guided optimization can be have a large impact on performance. One major challenge with applications like SURF is that its performance is highly data dependent. We use these features of SURF to illustrate how profiling can be used to characterize performance, identify bottlenecks, and explore optimization spaces. The main contributions of our work include:

- A new methodology for providing profiling for OpenCL parallel applications.
- A detailed description of this methodology when capturing OpenCL events.
- An application of this framework to accelerate a classical Computer Vision application.

This paper is organized as follows. In Section 2 we provide background on parallel program analysis, continuous profiling and *interest point detection* in Computer Vision. In Section 3, we briefly describe the SURF algorithm and discuss appropriate methods for parallelization. In Section 4 we discuss our event handling framework. In Section 5 we cover potential usage scenarios of our framework using SURF.

2. BACKGROUND AND RELATED WORK

2.1 Heterogeneous Application Profiling

Analysis and profiling of heterogeneous applications is a burgeoning research topic. There has been significant previous work discussing how best to measure performance of heterogeneous systems. Many of these studies focused on individual kernel optimization [25, 22]. If they do consider multi-kernel profiling, they require device driver modifications and external libraries. For example, Malony et al. developed TAUcuda, which can profile CUDA applications using the TAU Performance System [18]. Our work follows a similar path, but uses higher level function calls as provided for in the OpenCL specification. While we cannot obtain low-level information available specific to a driver/device, we apply our framework across different devices and can utilize it for a range of purposes including automated tuning and assessment of kernel optimizations.

Continuous runtime profiling has been studied for identifying application stalls in commercial data-centers [3] where the benefits of providing a built-in profiling subsystem are well established. Runtime profiling for multicore architectures is discussed by Furlinger et al. [13]. This prior work deals with runtime profiling in OpenMP, and requires annotation of source code. Spafford et al. also take a multi-kernel approach to profiling [23], but utilize the external Tau system for their analysis.

2.2 OpenCL Profiling

The OpenCL standard includes a profiling function `clGetEventProfilingInfo`, which returns timing statistics regarding OpenCL commands such as kernel launch and device IO. Common usage scenarios of OpenCL events include managing asynchronous IO and timing of code [19, 14].

Previous work on scheduling algorithms targeting heterogeneous devices [16] is complimentary to our work. A profiling framework that binds tightly with an open standard can enable more complicated and dynamic scheduling policies across devices for programs whose performance is data-driven. We aim to be able to extract profiling data from an application like Tau, and across platforms and devices like gDebugger [21] which is a debugging framework for GPUs.

2.3 Computer Vision and Interest Point Detection

Interest Point Detection (IPD) in digital imagery is a classic Computer Vision task. Over the past decade, a number of novel IPD algorithms and methods have been developed [4, 26]. Some of the target applications that use IPD include point matching to enable panoramic stitching and face identification/matching for a range of security applications [5, 17]. For many of the algorithms in IPD, parallelization can play an important role, especially as the number of images to be analyzed and the image resolution increases rapidly. Parallelization can be applied to real-time applications such as video stabilization, where a parallel implementation is needed in order to meet the real-time frame rate deadline.

The Computer Vision community has embraced heterogeneous computing because of the large number of applications in the field that can potentially benefit from parallelization [8]. For example, SURF is one Computer Vision application that possesses significant opportunities for parallelization, and a number of developers have already produced parallel implementations [28, 27, 12]. As a demonstration of the value of our new profiling infrastructure for OpenCL, we have taken a CUDA implemention of SURF [9], migrated it to OpenCL and added additional OpenCL kernels to the library. We call our new framework *Parallel SURF*.

3. PARALLEL SURF

The SURF application was first described by Bay et al. in 2006 [4]. SURF analyzes an image and produces feature vectors for points of interest (“ipoints”). SURF features have been used to perform operations like object recognition [17], feature comparison [27],face recognition [10]. A feature vector describes a set of ipoints. An ipoint consists of:

- The location of the point in the image.
- The local orientation at the detected point.
- The scale at which the interest point was detected.
- A descriptor vector(typically 64 values), that can be used to compare with the descriptors of other features.

The application is summarized in Figure 1 and we refer the interested reader to [4, 11] for a detailed discussion. Numerous projects are available on the Internet that have implemented elements of SURF in parallel using OpenMP [24] and CUDA [28, 27, 12]. The primary kernels in SURF are summarized in Table 1. We present the first (to our knowledge) parallelized implementation of SURF in OpenCL.

To find points of interest, SURF applies a *Fast-Hessian Detector* that uses approximated Gaussian Filters at different scales to generate a stack of Hessian matrices. SURF

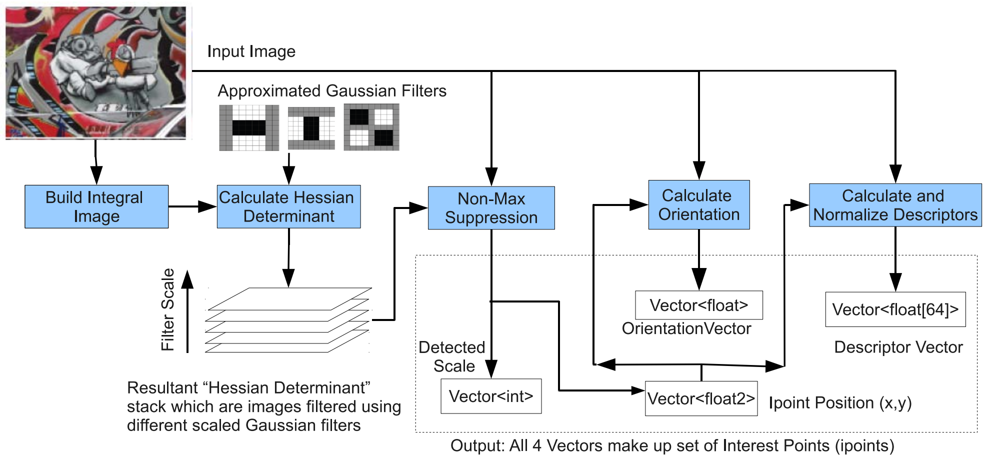

Input Image
Approximated Gaussian Filters
Build Integral Image
Calculate Hessian Determinant
Non-Max Suppression
Calculate Orientation
Calculate and Normalize Descriptors
Filter Scale
Vector<float>
OrientationVector
Vector<float[64]>
Descriptor Vector
Detected Scale
Vector<int>
Vector<float2>
Ipoint Position (x,y)
Resultant "Hessian Determinant" stack which are images filtered using different scaled Gaussian filters
Output: All 4 Vectors make up set of Interest Points (ipoints)

Figure 1: The Surf Algorithm performs convolution using Gaussian filters at different scales to build a stack of images. The stack of images are used to calculate the location of an ipoint. 64 element descriptors are calculated for each ipoint. Kernels within each block are presented in Table 1.

utilizes an *integral image* [15], which allows us to scale the filter instead of the image. The location of the ipoint is calculated by finding the local maxima or minima in the image at different scales using the generated Hessian matrices.

The local orientation at an ipoint allows us to maintain invariance to image rotation. The local orientation (4th stage of the pipeline in Figure 1) is calculated using the wavelet response in the X and Y directions in the neighborhood of the detected ipoint. The dominant local orientation is selected by rotating a circle segment covering an angle of $\pi/3$ around the origin. At each position, the x and y-responses within the segment of the circle are summed and used to form a new vector. The orientation of the longest vector becomes the feature orientation. The calculation of the largest response is done using a local memory-based reduction.

The 64 element descriptor is calculated by dividing the neighborhood of the ipoint into 16 regular subregions. Haar wavelets are calculated in each region and each region contributes 4 values to the descriptor. Thus, $16 * 4$ values are used in applications based on SURF to compare descriptors.

For the implementation evaluated in this paper, we started from the OpenSURF library [11, 9], rewrote it in OpenCL and added additional OpenCL kernels. Our goal was to extract as much parallelism out of the framework as possible. In SURF, execution performance is determined by the characteristics of the data set rather than the size of the data. This is because the number of ipoints detected in the *non-max suppression* stage of the algorithm helps to determine the workgroup dimensions for the orientation and descriptor kernels. Computer Vision frameworks like SURF also have a large number of tunable parameters (e.g., a detection threshold, which changes the number of points detected in the suppression stage) which greatly impacts the performance of an application.

4. APPLICATION ANALYSIS METHODOLOGY

A profiling framework was built to carefully characterize the performance of our implementation of SURF when embedded in other Computer Vision applications. Our profiling infrastructure and approach are applicable to any many-kernel parallel application written in OpenCL. We break down execution based of OpenCL events. In this section, we provide an overview on OpenCL events and then describe our framework for recording events in a heterogeneous application.

4.1 OpenCL Events

To provide for coarse-grained synchronization, OpenCL provides barriers like `clFlush()` and `clFinish()`. To enforce finer-grained synchronization, the OpenCL specification provides an object known as an *event* which is used to determine the status of a command. Event objects identify unique commands in a queue and thus provide command-level control. *Wait lists* are arrays of which events that can be passed to OpenCL commands. Wait lists can be used to block on more than one command. Events and wait lists can also be used to enforce synchronization by checking command status in a multi-device usage model where the command queues may be capable of out-of-order execution, allowing devices to start executing commands as soon as possible.

The OpenCL standard includes a profiling function `clGetEventProfilingInfo()`, which returns timing statistics regarding OpenCL commands in a queue. We utilize this function heavily in our profiling framework for SURF developed in OpenCL. OpenCL events return time-stamps[1] and are associated with the following states of a command.

- **CL_QUEUED:** Command enqueued, waiting to be submitted
- **CL_SUBMITTED:** Command submitted to device
- **CL_START:** Command started executing on device
- **CL_END:** Command has finished execution

Timestamps can be used for execution time profiling, and combined with event wait lists, can allow us to better un-

[1]The OpenCL 1.1 specification provides time-stamps in nanoseconds for all conformant devices

Kernel Name	Kernel Function
Integral Row	Integral image across all rows
Integral Col	Integral image across all columns
Init-Det	Initialize Memory Allocated for stack of Images
Build-Det	Builds a stack of images after convolution of different scales
Nonmax Suppression	Calculate ipoint locations and scale by looking for outliers through image stack
GetOrientation	Calculate orientation with largest Haar response around ipoint
Surf 64	Calculate descriptor vector for image
Norm 64	Normalize the calculated descriptors for each ipoint

Table 1: List of the kernels implemented in OpenCL as part of SURF.

derstand the actual execution of commands through a command queue. OpenCL 1.1 [19] provides more functionality for events which can increase the utility of our work. OpenCL 1.1 also provides *user events* which have a similar interface to cl_event types, but that can be triggered by the user. The new specification also provides event callbacks that can be used to enqueue new commands.

4.2 Measurement Framework

Data values can have a dramatic impact on the performance of applications developed with SURF (as discussed in Section 5.3.2). SURF is presently used in a large range of settings. These factors have helped motivate our work on a whole-application profiling framework that assists us with tuning SURF, as well as any other OpenCL application. We would like to provide for online, *continuous* profiling, but the overhead associated with continuous measurements presents challenges [3, 13]. To profile an OpenCL application, a user would need to create events explicitly and match the occurence of each event with its location in the application using metadata. Run-time profiling frameworks are commonly vendor specific, require running applications multiple times [2, 20] and tend to only support legacy parallel computing environments [13].

Faced with the asynchronous execution nature of OpenCL commands, to safely gather profiling information we would need to introduce an explicit `clFinish()` command. Inserting `clFlush()` or `clFinish()` to query an OpenCL event (especially on the critical path of a program) would change the behavior of the application. The query would break the proper sequence of OpenCL commands executed by the underlying driver/runtime.

To avoid this problem, we have implemented an event handling framework, as shown in Figure 2. When the SURF application is launched, a unique index for each OpenCL command is passed to our framework. If the index has not been seen by the event request system, an event is allocated within our vector. Using this index, a pointer to a cl_event is returned to the enqueue function. The `cl_event` is populated by the OpenCL runtime.

After execution completes on the current SURF frame, the events can be queried. The management framework allows us to control the number of events allocated (our framework does not allocate more than one event per OpenCL command). This minimizes memory overhead because algorithms like SURF are typically run on large numbers of video / camera frames, and allocating new events for each frame is infeasible. We thus manage to record events without interrupting the SURF pipeline.

5. PERFORMANCE ANALYSIS

As discussed in Section 3, it is challenging to predict performance of an algorithm like SURF without knowing both the properties of the input data set and input parameters to the application (e.g., suppression threshold[2] and depth of the Hessian stack[3]). Computer Vision benchmarks [26] are useful if our goal is to tune individual kernels, but one of our future goals is to build a performance analysis framework generic enough to be used with other heterogeneous applications, so we pursue a different approach. To study performance, we use sample videos [7] to study the performance of SURF using our event handling framework. We test our framework on three state-of-the-art devices, as shown in Table 2.

5.1 Implementation Details

The aim of our work is to discuss performance of a data driven application like SURF and how a profiling framework can help develop insight into its characteristics at runtime. We do not aim to cover platform specific optimizations for the kernels in the SURF algorithm. Our goal while implementing the kernels of SURF was that our application should run correctly and efficiently on any platform without any changes.

The main kernels in SURF have been discussed in Section 3. We apply well-known GPU programming optimizations [9, 15, 22, 25], such as local memory utilization, and work group optimization. Such optimizations benefit both AMD and NVIDIA GPUs.

5.2 Profiling Overhead

As mentioned in Section 4.2, our traces are gathered online and saved as the application runs. We query the events after a frame is fully completed[4]. We avoid adding synchronization to the application. We allocate events only on the first pass through the SURF pipeline (see Figure 1).

We measure the overhead for the extra OpenCL events by running SURF on different devices with and without profiling. To disable profiling, event pointers returned to the `clEnqueue` commands are set to NULL and profiling is disabled for the command queue. End-to-end performance of SURF algorithm is measured per frame, using host based measurements. Figure 3 shows the average time per frame

[2]Dictates the number of detected features.

[3]Determines the number of convolutions and their shape in BuildDet.

[4]The granularity of querying events is left tunable to enable applications where per frame profiling may not be necessary.

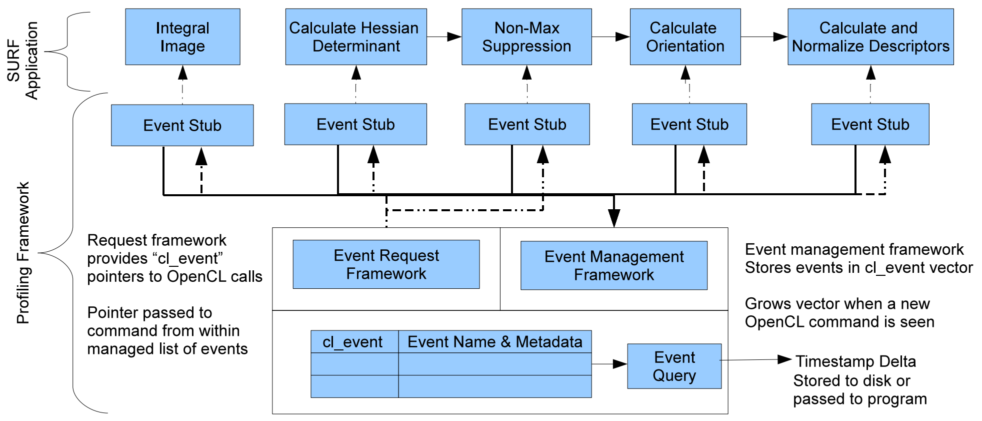

SURF Application
Integral Image
Calculate Hessian Determinant
Non-Max Suppression
Calculate Orientation
Calculate and Normalize Descriptors
Event Stub
Profiling Framework
Request framework provides "cl_event" pointers to OpenCL calls
Pointer passed to command from within managed list of events
Event Request Framework
Event Management Framework
cl_event
Event Name & Metadata
Event Query
Event management framework Stores events in cl_event vector
Grows vector when a new OpenCL command is seen
Timestamp Delta Stored to disk or passed to program

Figure 2: The OpenCL event profiling framework. The event stub is used to assign an event from within a array of event objects to each runtime function invoked. The event management system records the profiler data received from the OpenCL runtime.

Platform	**Device Type**	**Features**	**Memory(GB)**	**Core Frequency (GHz)**
AMD Phenom II x6	CPU	6 cores	4	3.79
AMD 5870	GPU	1600 SPUs	1	0.850
NVIDIA GTX480 (Fermi)	GPU	480 CUDA Cores	3	1.4

Table 2: The devices tested with OpenCl OpenSURF.

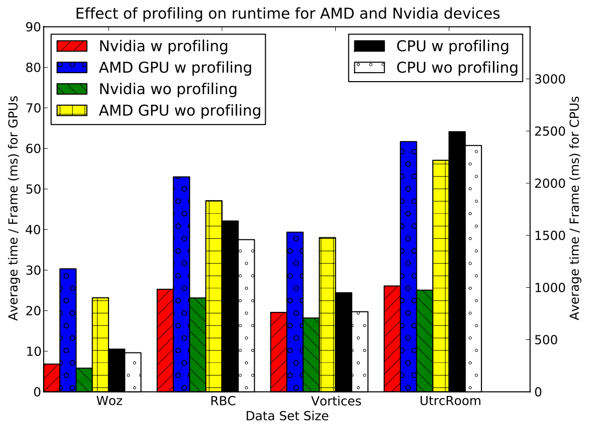


Figure 3: Per frame execution times for different data-sets, with profiling enabled and disabled.

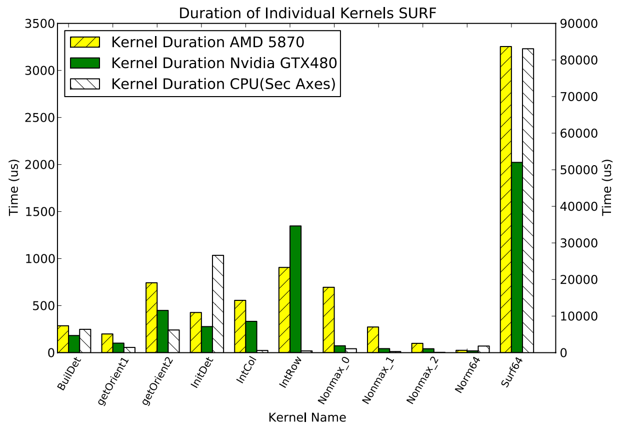


Figure 4: Kernel execution duration for different GPU devices. BuildDet has smaller execution time per kernel, but is typically called an order of magnitude more times than other kernels.

for each data set. As we can see, there is only a minimal change in performance when the profiling is enabled on AMD and NVIDIA GPUs.

Movie Name	**Size**
Woz	320 x 240
Voronoi	300 x 300
Vortices	687 x 611
RBC	791 x 704
UTRC	791 x 704

Table 3: Details of video samples used [7] for analysis.

5.3 Profiling Usage

The benefits of continuously profiling an application have been discussed previously [3, 13]. We briefly describe how the performance of SURF can be better understood using a profiling framework.

5.3.1 Guiding Optimization Steps

In any profile-guided optimization work, and especially when working with heterogeneous computing systems, one or a few kernels or functions typical dominate performance. It is pretty obvious that we should optimize these kernels first. In previous studies [15, 25, 22], conventional timing profiles have been used to guide a user as to which kernel should be optimized.

However, applications like SURF have many kernels with comparable execution times. This may also occur when a subset of kernels are called multiple times. This complicates the decision of where optimization efforts should be invested. A runtime profile of different kernels in SURF are shown in Figure 4. The BuildDet kernel is called once to build each image in intermediate output stack (Figure 1). The stack consists of at least 12 images [9, 11]. Surf64 is only called once per frame. As shown in Figure 5, the cumulative time spent in BuildDet is similar in duration to the time spent by Surf64. Thus, optimizing the simpler BuildDet kernel, and reducing the number of calls to BuildDet may be as beneficial as working on optimizing the more complicated Surf64 kernel.

In our implementation, the number of ipoints detected is read back to determine the workgroup dimensions for the subsequent steps. An easy optimization (as suggested in the NVIDIA programming guide [1]) would be to remove this I/O by hardwiring dimensions of later workgroups using approximate upper bounds. However, by looking at the flow of events in Figure 5b, we see that the transfer overhead of a single value is insignificant and the kernels executed later in the pipeline (i.e., orientation and Surf64) are the longest running kernels in our algorithm. So we decided that it was not advisable to increase their work group size further.

5.3.2 *Data Dependant Application Performance*

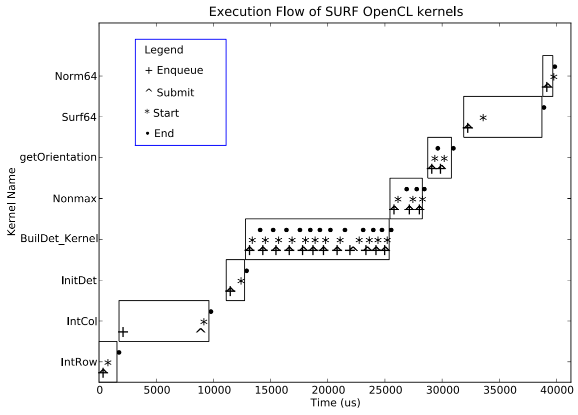
While profiling SURF execution, we record events on a per frame basis. The time per frame varies substantially, as seen from the timing values shown in Figure 6. This is due to the asynchronous nature of execution and variations in computation generated by the varying number of ipoints across frames(shown on secondary axes in Figure 6). By averaging results from GPU events across multiple frames into timelines (as shown in Figure 5), we can build a consistent and reproducible performance profile for a dataset.

Applications like feature matching use SURF as one of the primary computation kernels. By observing the number of features, input parameters and a *performance profile* for a data set, we can then use this information to improve performance of such computer vision applications. This can help guide optimizations of specific kernels based on the dataset and the target device. The tight coupling between the programming standard and the profiling framework permits us to use results from profiling to improve application performance.

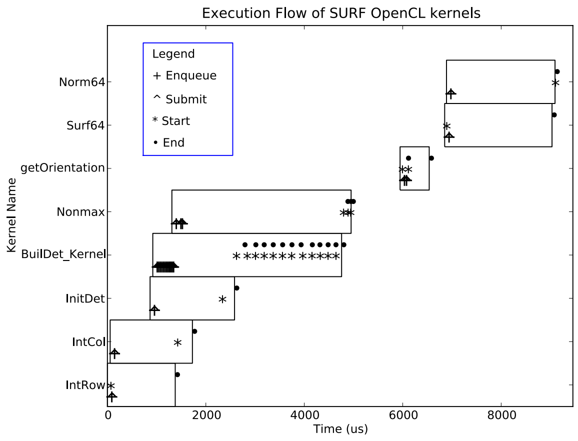
5.3.3 *Command Queue Behavior*

Figure 7 shows the usage of the command queue for the application. As we can see, when using either AMD or NVIDIA GPUs, the wait time of commands increases, as shown in Figure 7. We have plenty of kernels in our present implementation that can exploit the parallel acceleration provided by GPUs. We see similar characteristics in Figure 5b, where all the kernels are submitted by the CPU within 1 millisecond of the start, after which the CPU is idle till the orientation kernels are enqueued.

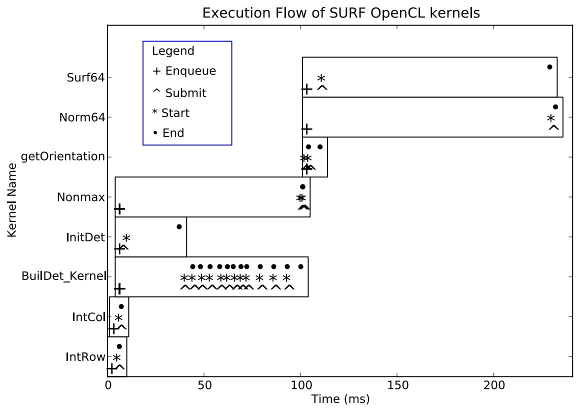
Figure 5b shows that we have pipelined the entire algorithm to the GPU. However, to further improve system efficiency, we would want to enqueue I/O while computation is in progress. From Figure 7, the best time to enqueue data (e.g., the next frame to extract features from) would be either at the beginning or before the start of the orientation



(a) AMD 5870 Timeline. The AMD GPU is run with synchronization for comparison purposes. Time/frame for the pipeline without synchronization is in Figure 3.



(b) Nvidia GTX480 Timeline.



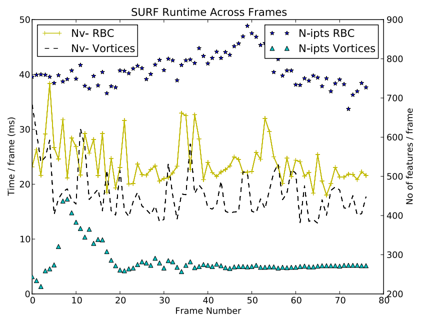
(c) AMD multicore CPU Timeline. Note that time is in milliseconds

Figure 5: Execution timeline of SURF kernels for different devices for the rbc video.

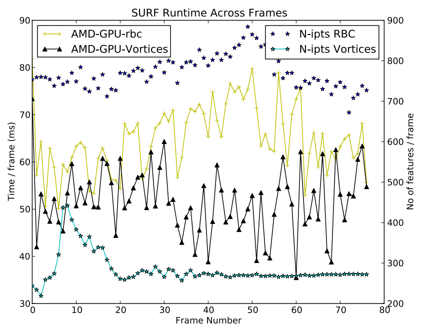
kernels. This should not distract the CPU from enqueueing computations, while allowing for a maximum overlap.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have demonstrated the value of utilizing



(a) Timing variation per frame- Nvidia



(b) Timing variation per frame - AMD GPUs

Figure 6: Variation of execution time per frame.

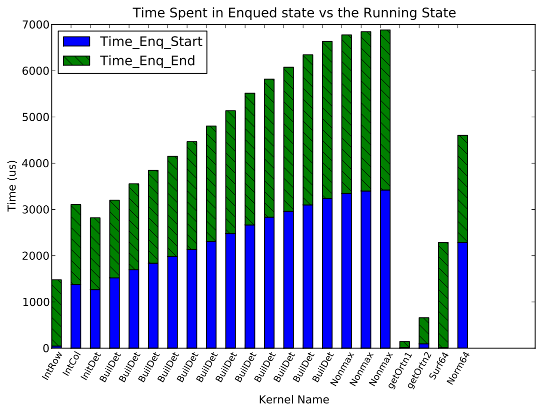


Figure 7: Breakdown of time spent by the OpenCL kernels spent in different states in the command queue using the Nvidia GPU.

application profiling in a parallel programming environment. Given the complexity of current and future heterogeneous computing environments, we need tools that can help to identify performance bottlenecks and identify opportunities for whole-application optimization.

Our future work includes testing our implementation of SURF with Computer Vision applications such as facial detection and searching. We also plan to test our profiling framework with other heterogeneous applications including multi-physics simulators and biomedical imaging. We anticipate the availability of more detailed profiling information in future versions of the OpenCL specification will greatly aid in our ability to reap maximal performance on these platforms.

Heterogeneous systems will continue to evolve; examples include using multiple GPUs on a system to run general purpose code, and AMD's Fusion technology, which will integrate a CPU and a GPU into one package. Opportunities exist for further acceleration of algorithms such as SURF; in the case of multiple GPUs on a system, it would be beneficial to pipeline the SURF kernels to maximize interest point detection throughput. Profiling on such a system could be accomplished with only minor extensions to the techniques described in this paper.

7. ACKNOWLEDGMENTS

The authors would like to thank Timothy Smith and Budirijanto Purnomo for their help in understanding the SURF algorithm and GPU profiling tools. The work presented in this paper was supported in part by the NSF through an EEC Innovation Award (EEC-0946463), by AMD through the AMD Strategic Academic Partners Program and by support of the Vice Provost's Office of Research at Northeastern University.

References

[1] CUDA programming Guide, version 2.0. *NVIDIA Corporation.*

[2] Cuda Visual Profiler. *NVIDIA Corporation.*

[3] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15:357–390, November 1997.

[4] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. *Computer Vision–ECCV 2006*, 2006.

[5] M. Brown and D. Lowe. Automatic panoramic image stitching using invariant features. *International Journal of Computer Vision*, 74(1):59–73, 2007.

[6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, page 786. ACM, 2004.

[7] J. Burkardt. Example avi files. World Wide Web.

[8] T. Chen, D. Budnikov, C. Hughes, and Y.-K. Chen. Computer vision on multi-core processors: Articulated body tracking. In *Multimedia and Expo, 2007 IEEE International Conference on*, pages 1862 –1865, 2-5 2007.

[9] M. Cowgill. Opensurf gpu enhancement. World Wide Web, 2009.

[10] G. Du, F. Su, and A. Cai. Face recognition using SURF features. In *Proc. of SPIE Vol*, volume 7496, pages 749628–1, 2009.

[11] C. Evans. Notes on the opensurf library. *University of Bristol, Tech. Rep. CSTR-09-001, January*, 2009.

[12] P. Furgale, C. Tong, and G. Kenway. ECE1724 Project Speeded-Up Speeded-Up Robust Features. 2009.

[13] K. Furlinger and S. Moore. Continuous runtime profiling of OpenMP applications. In *Proceedings of the International Conference on Parallel Computing (ParCo 07)(Advances in Parallel Computing*, volume 15.

[14] D. Gerstmann. Opencl event model usage. SIGGRAPH ASIA 2009.

[15] M. Harris, S. Sengupta, and J. Owens. Parallel prefix sum (scan) with CUDA. *GPU Gems*, 3(39):851–876, 2007.

[16] V. J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC '09, pages 19–33, Berlin, Heidelberg, 2009. Springer-Verlag.

[17] J. Luo, Y. Ma, E. Takikawa, S. Lao, M. Kawade, and B. Lu. Person-specific SIFT features for face recognition. In *IEEE International Conference on Acoustics, Speech and Signal Processing, 2007. ICASSP*, volume 2, 2007.

[18] A. D. Malony, S. Biersdorff, W. Spear, and S. Mayanglambam. An experimental approach to performance measurement of heterogeneous parallel applications using cuda. In *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*, pages 127–136, New York, NY, USA, 2010. ACM.

[19] A. Munshi. The OpenCL specification version 1.1. *Khronos OpenCL Working Group*, 2010.

[20] B. Purnomo, N. Rubin, and M. Houston. ATI Stream Profiler: a tool to optimize an OpenCL kernel on ATI Radeon GPUs. In *ACM SIGGRAPH Posters*. ACM, 2010.

[21] G. Remedy. 2010.

[22] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.

[23] K. Spafford, J. Meredith, J. Vetter, J. Chen, R. Grout, and R. Sankaran. Accelerating S3D: A GPGPU Case Study. In *Euro-Par 2009, Parallel Processing-Workshops. The Netherlands, August 25-28, 2009, Workshops*, page 122. Not Avail, 2010.

[24] S. Srinivasan, Z. Fang, R. Iyer, S. Zhang, M. Espig, D. Newell, D. Cermak, Y. Wu, I. Kozintsev, and H. Haussecker. Performance characterization and optimization of mobile augmented reality on handheld platforms. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 128–137. Citeseer, 2009.

[25] B. Sukhwani and M. C. Herbordt. Gpu acceleration of a production molecular docking code. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 19–27, New York, NY, USA, 2009. ACM.

[26] Venkata, S.K. and Ahn, I. and Donghwan Jeon and Gupta, A. and Louie, C. and Garcia, S. and Belongie, S. and Taylor, M.B. Sd-vbs: The san diego vision benchmark suite. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 55 –64, 2009.

[27] S. Warn, W. Emeneker, J. Gauch, J. Cothren, and A. Apon. Accelerating image feature comparisons using cuda on commodity hardware. Knoxville, TN, July 2010. Symposium on Application Accelerators in High Performance Computing (SAAHPC).

[28] N. Zhang. Computing Optimised Parallel Speeded-Up Robust Features (P-SURF) on Multi-Core Processors. *International Journal of Parallel Programming*, 38(2):138–158, 2010.