Maestro: Data Orchestration and Tuning for OpenCL Devices

Kyle Spafford, Jeremy Meredith, and Jeffrey Vetter

Oak Ridge National Laboratory
{spaffordkl,jsmeredith,vetter}@ornl.gov

Abstract. As heterogeneous computing platforms become more prevalent, the programmer must account for complex memory hierarchies in addition to the difficulties of parallel programming. OpenCL is an open standard for parallel computing that helps alleviate this difficulty by providing a portable set of abstractions for device memory hierarchies. However, OpenCL requires that the programmer explicitly controls data transfer and device synchronization, two tedious and error-prone tasks. This paper introduces Maestro, an open source library for data orchestration on OpenCL devices. Maestro provides automatic data transfer, task decomposition across multiple devices, and autotuning of dynamic execution parameters for some types of problems.

1 Introduction

In our previous work with general purpose computation on graphics processors (GPGPU) [1, 2], as well as a survey of similar research in the literature [3, 4, 5, 6, 7], we have encountered several recurring problems. First and foremost is code portability–most GPU programming environments have been proprietary, requiring code to be completely rewritten in order to run on a different vendor's GPU. With the introduction of OpenCL, the same kernel code (code which executes on the device) can generally be used on any platform, but must be "hand tuned" for each new device in order to achieve high performance. This manual optimization of code requires significant time, effort, and expert knowledge of the target accelerator's architecture.

Furthermore, the vast majority of results in GPGPU report performance for only single-GPU implementations, presumably due to the difficulty of task decomposition and load balancing, the process of breaking a problem into subtasks and dividing the work among multiple devices. These tasks require the programmer to know the relative processing capability of each device in order to appropriately partition the problem. If the load is poorly balanced, devices with insufficient work will be idle while waiting on those with larger portions of work to finish. Task decomposition also requires that the programmer carefully aggregate output data and perform device synchronization. This prevents OpenCL code from being portable–when moving to a platform with a different number of devices, or devices which differ in relative speed, work allocations must be adjusted.

Typically, GPUs or other compute accelerators are connected to the host processor via a bus. Many results reported in the literature focus solely on kernel execution and neglect to include the performance impact of data transfer across the bus. Poor management of this interconnection bus is the third common problem we have identified. The bandwidth of the bus is almost always lower than the memory bandwidth of the device, and suboptimal use of the bus can have drastic consequences for performance. In some cases, the difference in bandwidth can be more than an order of magnitude. Consider the popular NVIDIA Tesla C1060 which has a peak memory bandwidth of 102 gigabytes per second. It is usually connected to a host processor via a sixteen lane PCIe 2.0 bus, with peak bandwidth of only eight gigabytes per second.

One common approach to using the bus is the function offload model. In this model, sequential portions of an application execute on the host processor. When a parallel section is reached, input data is transferred to the accelerator. When the accelerator is finished computing, outputs are transferred back to the host processor. This approach is the simplest to program, but the worst case for performance. It poorly utilizes system resources–the bus is never active at the same time as the accelerator.

In order to help solve these problems, we have developed an open source library called Maestro. Maestro leverages a combination of autotuning, multi-buffering, and OpenCL's device interrogation capabilities in an attempt to provide a portable solution to these problems. Further, we argue that Maestro's automated approach is the only practical solution, since the parameter space for hand tuning OpenCL applications is enormous.

1.1 OpenCL

In December 2008, the Khronos Group introduced OpenCL [8], an open standard for parallel computing on heterogeneous platforms. OpenCL specifies a language, based on C99, that allows a programmer to write parallel functions called kernels which can execute on any OpenCL device, including CPUs, GPUs, or any other device with a supporting implementation. OpenCL provides support for data parallelism as well as task parallelism. OpenCL also provides a set of abstractions for device memory hierarchies and an API for controlling memory allocation and data transfer.

In OpenCL, parallel kernels are divided into tens of thousands of work items, which are organized into local work groups. For example, in matrix multiplication, a single work item might calculate one entry in the solution matrix, and a local work group might calculate a submatrix of the solution matrix.

2 Overview

Before proceeding to Maestro's proposed solutions to the observed problems, it is important to introduce one of the key ideas in Maestro's design philosophy, a single, high level task queue.

2.1 High Level Queue

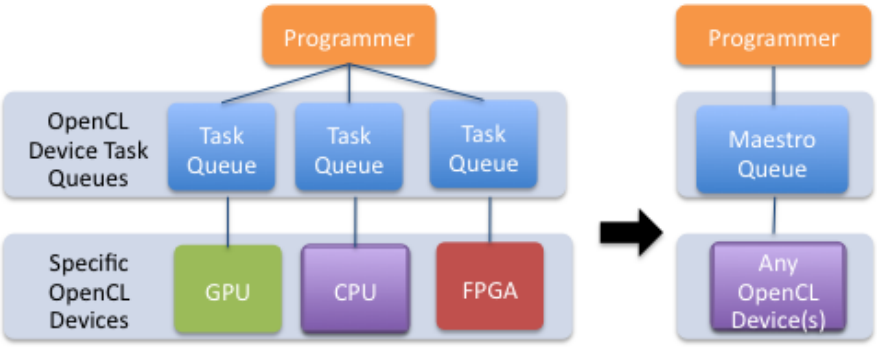

Fig. 1. Task Queue Hierarchies–In the OpenCL task queue hierarchy, the programmer must manage a separate task queue for each device. Maestro unifies these into a single, high-level queue which is independent of the underlying hardware.

In the OpenCL task queue model, the programmer must manage a separate task queue for each GPU, CPU, or other accelerator in a heterogeneous platform. This model requires that the programmer has detailed knowledge about which OpenCL devices are available. Modifications to the code are required to obtain high performance on a system with a different device configuration.

The Maestro model (contrasted in Figure 1), unifies the disparate, device-specific queues into a single, high-level task queue. At runtime, Maestro queries OpenCL to obtain information about the available GPUs or other accelerators in a given system. Based on this information, Maestro can transfer data and divide work among the available devices automatically. This frees the programmer from having to synchronize multiple devices and keep track of device specific information.

3 Problem: Code Portability

OpenCL's claim to portability relies on its ability to execute kernel code on any device with a supporting implementation. While this represents a substantial improvement over proprietary programming environments, some obstacles to portability remain.

One such obstacle is the organization of local work items. What is the appropriate local work group size for a given kernel? With current hardware, local work items roughly correspond to device threads. Hence, for GPUs, the rule of thumb is to start with a sufficiently high multiple of sixteen, e.g. 128 or 256. However, this heuristic does not guarantee a kernel will execute successfully,

much less exhibit high performance. For example, the OpenCL implementation in Mac OS X imposes an upper limit on local group size of *one* for code to execute on a CPU. Also, while larger group sizes often lead to better performance, if the kernel is strongly constrained by either register or local memory usage, it may simply fail to execute on a GPU due to lack of resources.

3.1 Proposed Solution: Autotuning

Since OpenCL can execute on devices which differ so radically in architecture and computational capabilities, it is difficult to develop simple heuristics with strong performance guarantees. Hence, Maestro's optimizations rely solely on empirical data, instead of any performance model or *a priori* knowledge. Maestro's general strategy for all optimizations can be summarized by the following steps:

1. Estimate based on benchmarks
2. Collect empirical data from execution
3. Optimize based on results
4. While performance continues to improve, repeat steps 2-3

This strategy is used to optimize a variety of parameters including local work group size, data transfer size, and the division of work among multiple devices. However, these dynamic execution parameters are only one of the obstacles to true portability. Another obstacle is the choice of hardware specific kernel optimizations. For instance, some kernel optimizations may result in excellent performance on a GPU, but reduce performance on a CPU. This remains an open problem. Since the solution will no doubt involve editing kernel source code, it is beyond the scope of Maestro.

4 Problem: Load Balancing

In order to effectively distribute a kernel among multiple OpenCL devices, a programmer must keep in mind, at a minimum, each device's relative performance on that kernel, the speed of the interconnection bus between host processor and each device (which can be asymmetric), a strategy for input data distribution to devices, and a scheme on how to synchronize devices and aggregate output data. Given that an application can have many kernels, which can very significantly in performance characteristics (bandwidth bound, compute bound, etc.), it quickly becomes impractical to tune optimal load balancing for every task by hand.

4.1 Proposed Solution: Benchmarks and Device Interrogation

At install time, Maestro uses benchmarks and the OpenCL device interrogation API to characterize a system. Peak FLOPS, device memory bandwidth, and bus bandwidth are measured using benchmarks based on the Scalable Heterogeneous Computing (SHOC) Benchmark Suite [9]. The results of these benchmarks serve

as the basis, or initial estimation, for the optimization of the distribution of work among multiple devices.

As a kernel is repeatedly executed, either in an application or Maestro's offline tuning methods, Maestro continues to optimize the distribution of work. After each iteration, Maestro computes the average rate at which each device completes work items, and updates a running, weighted average. This rate is specific to each device and kernel combination, and is a practical way to measure many interacting factors for performance. We examine the convergence to an optimal distribution of work in Section 6.2.

5 Problem: Suboptimal Use of Interconnection Bus

OpenCL devices are typically connected to the host processor via a relatively slow interconnection bus. With current hardware, this is normally the PCIe bus. Since the bandwidth of this bus is dramatically lower than a GPU's memory bandwidth, it introduces a nontrivial amount of overhead.

5.1 Proposed Solution: Multibuffering

In order to minimize this overhead, Maestro attempts to overlap computation and communication as much as possible. Maestro leverages and extends the traditional technique of double buffering (also known as ping-pong buffering).

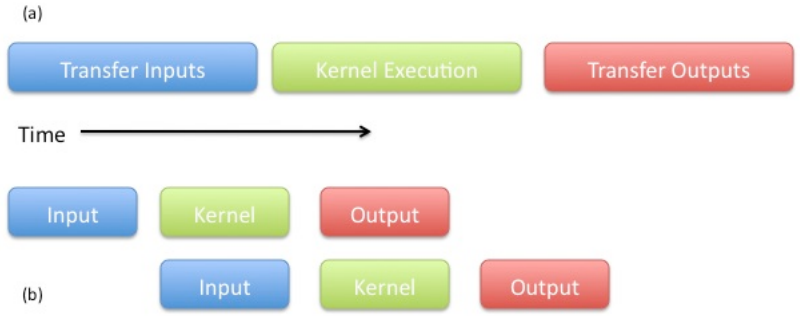

Fig. 2. Double Buffering–This figure contrasts the difference between (a) the function offload model and (b) a very simple case of double buffering. Devices which can concurrently execute kernels and transfer data are able to hide some communication time with computation.

Figure 2 illustrates the difference between the function offload model and double buffered execution. Maestro implements concurrent double buffering to multiple devices, including optimization of the data chunk size, which we term multibuffering.

In Maestro's implementation of multibuffering, the initial data chunk size is set to the size that resulted in the maximum bus bandwidth measured by benchmarks at install time. Maestro then varies the chunk size and optimizes based on observed performance.

However, double buffering cannot be used in all cases. Some OpenCL platforms simply lack the support for concurrent data copy and execution. Furthermore, some algorithms are not practical for use with double buffering. Consider an algorithm which accesses input data randomly. A work item might require data at the end of an input buffer which has not yet been transferred to the accelerator, resulting in an error. In order to accommodate this class of algorithms, Maestro allows the programmer to place certain inputs in a universal buffer, which is copied to all devices before execution begins. While this does limit the availability of some performance optimizations, it greatly expands the number of algorithms which can be supported by Maestro.

6 Results

6.1 Experimental Testbeds

OpenCL Limitations Since OpenCL is still a nascent technology, early software implementations impose several restrictions on the composition of test platforms. First, it is not possible to test a system with GPUs from different vendors due to driver and operating system compatibility issues. Second, CPU support is not widely available. As such, we attempt to provide results from a comprehensive selection of devices, including platforms with homogeneous GPUs, heterogeneous GPUs, and with an OpenCL-supported CPU and GPU.

Host Configurations

- **Krakow** Krakow is a dual socket Nehalem based system, with a total of eight cores running at 2.8Ghz with 24GB of RAM. Krakow also features an NVIDIA Tesla S1070, configured to use two Tesla T10 processors connected via a sixteen lane PCIe v2.0 bus. Results are measured using NVIDIA's GPU Computing SDK version 3.0.
- **Lens** Lens is a medium sized cluster primarily used for data visualization and analysis. Its thirty-two nodes are connected via Infiniband, with each node containing four AMD quad core Barcelona processors with 64GB of RAM. Each node also has two GPUs–one NVIDIA Tesla C1060 and one NVIDIA GeForce 8800GTX, connected to the host processor over a PCIe 1.0 bus with sixteen active lanes. Lens runs Scientific Linux 5.0, and results were measured using NVIDIA's GPU computing SDK, version 2.3.
- **Lyon** Lyon is an dual-socket, single-core 2.0 GHz AMD Opteron 246 system with a 16-lane PCIe 1.0 bus and 4GB of RAM, housing an ATI Radeon HD 5870 GPU. It runs Ubuntu 9.04 and uses the ATI Stream SDK 2.0 with the Catalyst 9.12 Hotfix 8.682.2RC1 driver.

Table 1. Comparison of Graphics Processors

GPU	Peak FLOPS	Mem. Bandwidth	Processors	Clock	Memory
Units	GF	GB/s	#	Mhz	MB
Tesla C1060/T10	933	102	240	1300	4096
GeForce 8800GTX	518	86	128	1350	768
Radeon HD5870	2720	153	1600	850	1024

Graphics Processors

- **NVIDIA G80 Series** The NVIDIA G80 architecture combined the vertex and pixel hardware pipelines of traditional graphics processors into a single category of cores, all of which could be tasked for general-purpose computation if desired. The NVIDIA 8800GTX has 128 processor cores split among sixteen multiprocessors. These cores run at 1.35GHz, and are fed from 768MB of GDDR3 RAM through a 384-bit bus.
- **NVIDIA GT200 Series** The NVIDIA Tesla C1060 graphics processor comprises thirty streaming multiprocessors, each of which contains eight stream processors for a total of 240 processor cores clocked at 1.3Ghz. Each multiprocessor has 16KB of shared memory, which can be accessed as quickly as a register under certain access patterns. The Tesla C1060 has 4GB of global memory and supplementary cached constant and texture memory.
- **ATI Evergreen Series** In ATI's "Terascale Graphics Engine" architecture, Stream processors are divided into groups of eighty, which are collectively known as SIMD cores. Each SIMD core contains four texture units, an L1 cache, and has its own control logic. SIMD cores can communicate with each other via an on-chip global data share. We present results from the Radeon HD5870 (Cypress XT) which has 1600 cores.

6.2 Test Kernels

We have selected the following five test kernels to evaluate Maestro. These kernels range in both complexity and performance characteristics. In all results, the same kernel code is used on each platform, although the problem size is varied. As such, cross-machine results are not directly comparable, and are instead presented in normalized form.

- **Vector Addition** The first test kernel is the simple addition of two one dimensional vectors, $C \leftarrow A + B$. This kernel is very simple and strongly bandwidth bound. Both input and output vectors can be multibuffered.
- **Synthetic FLOPS** The synthetic FLOPS kernel maintains the simplicity of vector addition, but adds in an extra constant, K, $C \leftarrow A + B + K$. K is computed using a sufficiently high number of floating point operations to make the kernel compute bound.

– **Vector Outer Product** The vector outer product kernel, $u \otimes v$, takes two input vectors of length n and m, and creates an output matrix of size $n \times m$. The outer product reads little input data compared to the generated output, and does not support multibuffering on any input.
– **Molecular Dynamics** The MD test kernel is a computation of the Lennard-Jones potential from molecular dynamics. It is a strongly compute bound, $O(n^2)$ algorithm, which must compare each pair of atoms to compute all contributions to the overall potential energy. It does not support multibuffering on all inputs.
– **S3D** We also present results from the key portion of S3D's Getrates kernel. S3D is a computational chemistry application optimized for GPUs in our previous work [2]. This kernel is technically compute bound, but also consumes seven inputs, making it the most balanced of the test kernels.

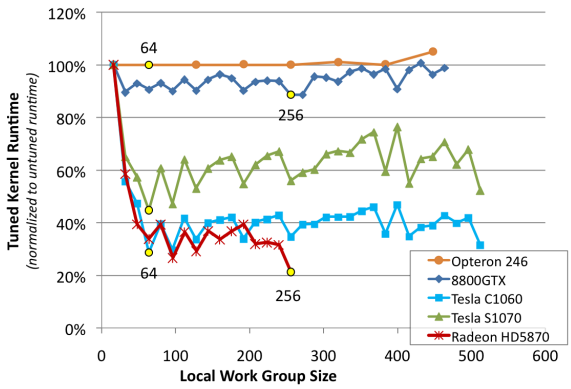

Fig. 3. Autotuning the local work group size – This figure shows the performance of the MD kernel on various platforms at different local work group sizes, normalized to the performance at a group size of 16. Lower runtimes are better.

Local Tuning Results Maestro's capability for autotuning of local work group size is shown using the MD kernel in Figure 3. All runtimes are shown in normalized fashion, in this case as a percentage of the runtime on each platform with a local work group size of 16 (the smallest allowable on several devices). The optimal local work group size is highlighted for each platform. Note the variability and unpredictability of performance due to the sometimes competing demands of register pressure, memory access patterns, and thread grouping. These results indicate that a programmer will be unlikely to consistently determine an optimal work group size at development time. By using Maestro's autotuning capability,

the developer can focus on writing the kernel code, not on the implications of local work group size on correctness and performance portability.

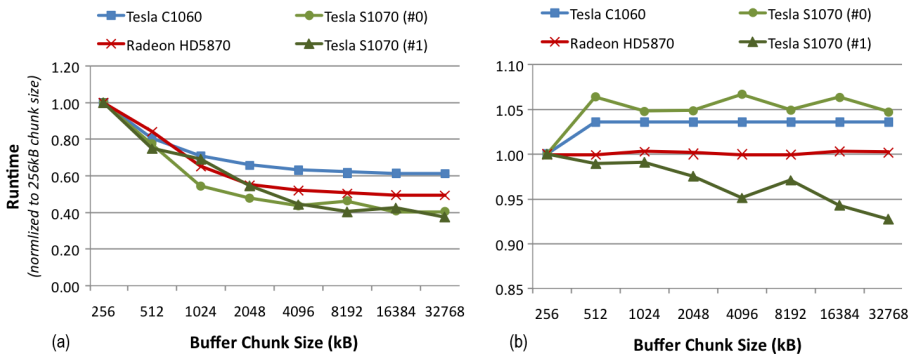

Fig. 4. Autotuning the buffer chunk size – This figure shows the performance on the (a) vector outer product and (b) S3D kernels when splitting the problem into various size chunks and using multibuffering. Lower runtimes are better. Values are normalized to the runtime at the 256kB chunk size on each platform.

Multibuffering Results Maestro's effectiveness when overlapping computation with communication can be improved by using an optimal buffer chunk size. Figure 4 shows Maestro's ability to auto-select the best buffer size on each platform. We observe in the vector outer product kernel one common situation, where the largest buffer size performs the best. Of course, the S3D kernel results show that this is not always the case; here, a smaller buffer size is generally better. However, note that on Krakow with two Tesla S1070 GPUs, there is an asymmetry between the two GPUs, with one preferring larger and one preferring smaller buffer sizes. This result was unusual enough to merit several repeated experiments for verification. Again, this shows the unpredictability of performance, even with what appears to be consistent hardware, and highlights the need for autotuning.

Mutli-GPU Results One of Maestro's strengths is its ability to automatically partition computation between multiple devices. To determine the proportion of work for each device, it initially uses an estimate based on benchmarks run at install time, but will quickly iterate to an improved load distribution based on the imbalance for a specific kernel. Figure 5 shows for the S3D and MD kernels the proportion of total time spent on each device. Note that there is generally an initial load imbalance which can be significant, and that even well balanced hardware is not immune. Maestro's ability to automatically detect and account for load imbalance makes efficient use of the resources available on any platform.

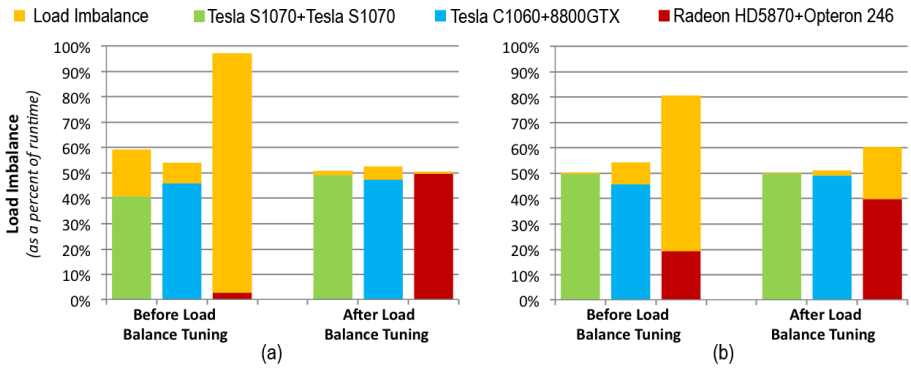

Fig. 5. Autotuning the load balance – This figure shows the load imbalance on the (a) S3D and (b) MD kernels, both before and after tuning the work distribution for the specific kernel. Longer yellow bars show a larger load imbalance.

Combined Results Maestro's autotuning has an offline and an online component. At install time, Maestro makes an initial guess for local work group size, buffering chunk size, and workload partitioning for all kernels based on values which are measured using benchmarks.

However, Maestro can do much better, running an autotuning process to optimize all of these factors, often resulting in significant improvements. Figure 6 shows the results of Maestro's autotuning for specific kernels relative to its initial estimate for these parameters. In (a) we see the single-GPU results, showing the combined speedup both from tuning the local work group size and applying double buffering with a tuned chunk size, showing improvement up to 1.60×. In (b) we see the multi-GPU results, showing the combined speedup both from tuning the local work group size and applying a tuned workload partitioning, showing speedups of up to 1.8×.

This autotuning can occur outside full application runs. Kernels of particular interest can be placed in a unit test and executed several times to provide Maestro with performance data (measured internally via OpenCL's event API) for coarse-grained adjustments. This step is not required, since the same optimizations can be performed online, but reduces the number of online kernel executions with suboptimal performance.

7 Related Work

An excellent overview of the history of GPGPU is given in [10]. Typically, work in this area has been primarily focused on case studies, which describe the process of accelerating applications or algorithms which require extremely high performance[3, 4, 5, 6, 7, 1, 2]. These applications are typically modified to use graphics processors, STI Cell, or field programmable gate arrays (FPGAs).

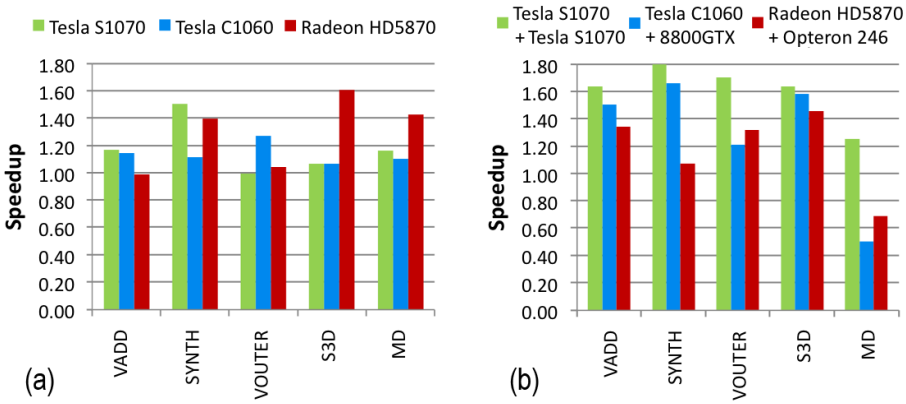

Fig. 6. Combined autotuning results – (a) Shows the combined benefit of autotuning both the local work group size the double buffering chunk size for a single GPU of the test platforms. (b) Shows the combined benefit of autotuning both the local work group size and the multi-GPU load imbalance using both devices (GPU+GPU or GPU+CPU) of the test platforms. Longer bars are better.

These studies serve as motivation for Maestro, as many of them help illustrate the aforementioned common problems.

Autotuning on GPU-based systems is beginning to gain some popularity. For example, Venkatasubramanian et. al. have explored autotuning stencil kernels for multi-CPU and multi-GPU environments [11]. Maestro is distinguished from this work because it uses autotuning for the optimization of data transfers and execution parameters, rather than the kernel code itself.

8 Conclusions

In this paper, we have presented Maestro, a library for data orchestration and tuning on OpenCL devices. We have shown a number of ways in which achieving the best performance, and sometimes even correctness, is a daunting task for programmers. For example, we showed that the choice of a viable, let alone optimal local work group size for OpenCL kernels cannot be accomplished with simple rules of thumb. We showed that multibuffering, a technique nontrivial to incorporate in OpenCL code, is further complicated by the problem- and device-specific nature of choosing an optimal buffer chunk size. And we showed that even in what appear to be well-balanced hardware configurations, load balancing between multiple GPUs can require careful division of the workload.

Combined, this leads to a space of performance and correctness parameters which is immense. By not only supporting double buffering and problem partitioning for existing OpenCL kernels, but also applying autotuning techniques to find the high performance areas of this parameter space with little developer

effort, Maestro leads to improved performance, improved program portability, and improved programmer productivity.

9 Acknowledgements

This manuscript has been authored by a contractor of the U.S. Government under Contract No. DE-AC05-00OR22725. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

References

[1] Meredith, J.S., Alvarez, G., Maier, T.A., Schulthess, T.C., Vetter, J.S.: Accuracy and Performance of Graphics Processors: A Quantum Monte Carlo Application Case Study. Parallel Computing **35**(3) (2009) 151–163

[2] Spafford, K.L., Meredith, J.S., Vetter, J.S., Chen, J., Grout, R., Sankaran, R.: Accelerating S3D: A GPGPU Case Study. In: HeteroPar '09: Proceedings of the Seventh International Workshop on Algorithms, Models, and Tools for Parallel Computing on Heterogeneous Platforms. (2009)

[3] Rodrigues, C.I., Hardy, D.J., Stone, J.E., Schulten, K., Hwu, W.M.W.: GPU Acceleration of Cutoff Pair Potentials for Molecular Modeling Applications. In: CF '08: Proceedings of the 2008 Conference on Computing Frontiers, New York, NY, USA, ACM (2008) 273–282

[4] He, B., Govindaraju, N.K., Luo, Q., Smith, B.: Efficient Gather and Scatter Operations on Graphics Processors. In: SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, New York, NY, USA, ACM (2007) 1–12

[5] Fujimoto, N.: Faster Matrix-Vector Multiplication on GeForce 8800GTX. IEEE International Symposium on Parallel and Distributed Processing, 2008 (IPDPS). (April 2008) 1–8

[6] Bolz, J., Farmer, I., Grinspun, E., Schröoder, P.: Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. In: ACM SIGGRAPH 2003, New York, NY, USA, ACM (2003) 917–924

[7] Stone, J.E., Phillips, J.C., Freddolino, P.L., Hardy, D.J., Trabuco, L.G., Schulten, K.: Accelerating Molecular Modeling Applications With Graphics Processors. Journal of Computational Chemistry **28** (January 2005) 2618–2640

[8] The Khronos Group: http://www.khronos.org/opencl/. (2009)

[9] Danalis, A., Marin, G., McCurdy, C., Mereidth, J., Roth, P., Spafford, K., Tipparaju, V., Vetter, J.: The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In: Proceedings of the Third Annual Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU '10), ACM (2010)

[10] Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J.: GPU Computing. Proceedings of the IEEE **96**(5) (May 2008) 879–899

[11] Venkatasubramanian, S., Vuduc, R.W.: Tuned and Wildly Asynchronous Stencil Kernels for Hybrid CPU/GPU Systems. In: ICS '09: Proceedings of the 23rd international conference on Supercomputing, New York, NY, USA, ACM (2009) 244–255