Automatic OpenCL Device Characterization: Guiding Optimized Kernel Design

Peter Thoman, Klaus Kofler, Heiko Studt,
John Thomson, and Thomas Fahringer

University of Innsbruck

Abstract. The OpenCL standard allows targeting a large variety of CPU, GPU and accelerator architectures using a single unified programming interface and language. While the standard guarantees portability of functionality for complying applications and platforms, performance portability on such a diverse set of hardware is limited. Devices may vary significantly in memory architecture as well as type, number and complexity of computational units. To characterize and compare the OpenCL performance of existing and future devices we propose a suite of microbenchmarks, uCLbench.

We present measurements for eight hardware architectures – four GPUs, three CPUs and one accelerator – and illustrate how the results accurately reflect unique characteristics of the respective platform. In addition to measuring quantities traditionally benchmarked on CPUs like arithmetic throughput or the bandwidth and latency of various address spaces, the suite also includes code designed to determine parameters unique to OpenCL like the dynamic branching penalties prevalent on GPUs. We demonstrate how our results can be used to guide algorithm design and optimization for any given platform on an example kernel that represents the key computation of a linear multigrid solver. Guided manual optimization of this kernel results in an average improvement of 61% across the eight platforms tested.

1 Introduction

The search for higher sustained performance and efficiency has, over recent years, led to increasing use of highly parallel architectures. This movement includes GPU computing, accelerator architectures like the Cell Broadband Engine, but also the increased thread- and core-level parallelism in classical CPUs [9]. In order to provide a unified programming environment capable of effectively targeting this variety of devices, the Khronos group proposed the OpenCL standard. It includes a runtime API to facilitate communication with devices and a C99-based language specification for writing device code. Currently, many hardware vendors provide implementations of the standard, including AMD, NVIDIA and IBM.

The platform model for OpenCL comprises a *host* – the main computer – and several *devices* featuring individual *global memory.* Computation is performed by invoking data-parallel *kernels* on an N-dimensional grid of *work items*. Each point in the grid is mapped to a *processing element*, and elements are grouped in *compute units* sharing *local memory*. Broad acceptance of the standard leads to the interesting situation where vastly different hardware architectures can be targeted with essentially unchanged code. However, implementations suited well to one platform may – because of seemingly small architectural differences – fail to perform acceptably on other platforms. The large and increasing number of hardware and software targets and complex relationships between code and performance changes make it hard to gain an understanding of how some algorithm will perform across the full range of platforms.

In order to enable automated in-depth characterization and comparison of OpenCL hardware and software platforms we have created a suite of microbenchmarks – uCLbench. It provides programs measuring the following data points:

Arithmetic Throughput. Parallel and sequential throughput for all basic mathematical operations, and many built-in functions defined by the OpenCL standard. When available, native implementations (with reduced accuracy) are also measured.

Memory Subsystem. Host to device, device to device and device to host copying bandwidth. Streaming bandwidth for on-device address spaces. Latency for memory accesses to global, local and constant address spaces. Also determines existence and size of caches.

Branching Penalty. Impact of divergent dynamic branching on device performance, particularly pronounced on GPUs.

Runtime Overheads. Kernel compilation time and queuing delays incurred when invoking kernels of various code volume.

2 Benchmark Design and Methodology

Before examining the individual benchmarks composing the uCLbench suite the basic goals that shaped our design decisions need to be established. The primary purpose of the suite is to characterize and compare the low-level performance of OpenCL devices and implementations. As such, we did not employ device-specific workarounds to ameliorate problems affecting performance on some particular device, since the same behavior would be encountered by actual programs. Another concern is providing implementers with useful information that can support them in achieving good performance over a broad range of devices. Particularly the latency and branching penalty benchmarks are designed with this goal in mind.

There are three main implementation challenges for uCLbench:

1. **Ensure accuracy.** The benchmarks need to actually measure the intended quantity on all devices tested, and it must be possible to verify the computations performed.
2. **Minimize overheads.** Overheads are always a concern in microbenchmarks, but with the variety of devices available to OpenCL they can be

hard to avoid. E.g. a simple loop that is negligible in its performance impact on a general purpose CPU can easily dominate completion time on a GPU.

3. **Prevent compiler optimization.** Since kernel code is compiled at runtime using the compiler provided by the OpenCL implementation, we have no control over the generated code. Thus it is imperative to design the benchmarks in a way that does not allow the compiler to perform unintended optimizations. Such optimizations could result in the removal of operations that should be measured.

There is an obvious area of conflict between these three goals. It is particularly challenging to prevent compiler optimization while not creating significant overheads that could compromise accuracy – even more so when the same code base is used on greatly differing hardware and compiled by different closed-source optimizing compilers.

2.1 Arithmetic Throughput

As a central part of the suite, this benchmark measures the arithmetic capabilities of a device. It includes primitive operations as well as many of the complex functions defined in the OpenCL standard. Two distinct quantities are determined: the device-wide throughput that can be achieved by independent parallel execution as well as the performance achieved for sequentially dependent code. All measurements are taken for scalar and vector types, and, if available, both native (less accurate) and default versions of complex functions are considered.

To enable result checking and prevent compiler optimization, input and output are performed by means of global memory pointers, and the result of each operation is used in subsequent ones. The loop is manually unrolled to minimize loop overheads on all devices. Automatic unrolling can not be relied upon to achieve repeatable results for all platforms and data/operation types.

The kernel is invoked with a local and global range of one work item to determine the sequential time required for completion of the operation, and with a local range of loc = `CL_DEVICE_MAX_WORK_GROUP_SIZE` and a global range of `CL_DEVICE_MAX_COMPUTE_UNITS`$*loc$ items to calculate device-wide throughput.

2.2 Memory Subsystem

Current GPUs and accelerator devices have a memory design that differs from the deep cache hierarchies common in CPUs.

Bandwidth. While global GPU memory bandwidth per-chip is high, due to the degree of hardware parallelism the memory bandwidth available per processing element can be insufficient [13]. Another bottleneck for current GPUs is the PCIe slot intermediating host memory and device.

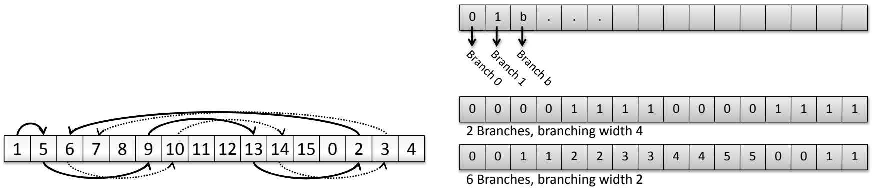
Many GPUs and accelerators attempt to ameliorate these issues by providing a manually controlled, multi-layered memory subsystem. In OpenCL, this concept is represented by separate address spaces: private, local, constant, global and host memory.

For this reason, the benchmark is divided in two major parts: one for on-device memory layers and one for memory traffic between host memory and device. To test bandwidth for on-device memory the benchmark invokes kernels streaming data from one layer back into the same layer. We also discern differences between scalar and various vectorized types, as the latter might be optimized.

Host $\leftrightarrow$ device bandwidth measurement does not require any kernel, instead it uses the runtime API for copying data from/to the device's global memory or inside device global memory. For device/host communication, two options are considered: the first generates a buffer and commands the OpenCL runtime to transfer it (`clEnqueueWriteBuffer`), the second *maps* a device buffer into the host memory and works directly on the returned pointer.

For the streaming kernel, overheads were a major concern. This was addressed by using fast add operations to forestall optimization, and by maximizing the ratio of read/write memory accesses.

Latency. In addition to bandwidth, knowledge about access latency is essential to effectively utilize the available OpenCL memory spaces. Depending on the device used, only some or none of the accesses may be cached, and latency can vary by two orders of magnitude, from a few cycles up to several hundreds.



(a) Latency benchmark offset array for a cache line size of 4 elements (b) Branch penalty measurement layout options

Fig. 1. Patterns used for latency and branch penalty benchmark, respectively

The latency benchmark uses a specifically designed index array to perform a large number of indirect memory accesses. The index array contains address offsets chosen to cause jumps larger than cache line size, and end on a zero entry after traversing the entire array, as illustrated in Fig. 1(a).

Some input-dependent computation and output has to be performed to prevent optimization, which is achieved by accumulating offsets. Manual loop unrolling is used to minimize overheads. When measuring local memory latency a large number of repeated traversals is required.

2.3 Branching Penalty

On some OpenCL devices divergent dynamic branching on work items leads to some or all work being serialized. The impact can differ with the amount and topological layout of diverging branches on the work range. Since the effect on algorithm performance of this penalty can be severe [6] we designed a microbenchmark to determine how devices react to various branch counts and layouts.

The benchmark kernel is provided with an array of floating point numbers equal in length to the amount of work items. Each item then takes a branch depending on the number stored in its assigned location. Fig. 1(b) illustrates how `brancharray` configurations can be used to test a varying number of branches and different branch layouts.

2.4 Runtime Overheads

Compared to traditional program execution, the OpenCL model introduces two potential sources of overhead. Firstly, it is possible to compile kernels at runtime, and secondly there is an amount of time spent between queuing a kernel invocation and the start of computation. These overheads are measured in uCLBench using the OpenCL profiling event mechanism – we define the invocation overhead as the elapsed time between the `CL_PROFILING_COMMAND_QUEUED` and `CL_PROFILING_COMMAND_START` events, and the compilation time as the time spent in the `clBuildProgram` call. The actual kernel execution time is disregarded for this benchmark, and the accuracy of the profiling events is implementation defined (see Table 1).

3 Device Characterization – Results

To represent the broad spectrum of OpenCL-capable hardware we selected eight devices, comprising four GPUs, three CPUs and one accelerator. Their device characteristics as reported by OpenCL are summarized in Table 1.

NVIDIA TESLA 2050. The GF100 Fermi chip in this GPGPU device contains 14 compute units with a load/store unit, a cluster of four special function units as well as two clusters of 16 scalar processors each. The scalar processor

Table 1. OpenCL devices benchmarked

Device	Tesla2050	Radeon5870	GTX460	GTX275	2x X5570	2x Opt.2435	2xCellPPE	2xCellSPE
Implementation	NVIDIA	AMD	NVIDIA	NVIDIA	AMD	AMD	IBM	IBM
Operating System	CentOS5.3	CentOS5.4	CentOS5.4	Win 7	CentOS5.4	CentOS5.4	YDL 6.2	YDL 6.2
Host Connection	PCIe 2.0	PCIe 2.0	PCIe 2.0	PCIe 2.0	-	-	-	On-chip
Type	GPU	GPU	GPU	GPU	CPU	CPU	CPU	ACCEL
Compute Units	14	20	7	30	16	12	4	16
Max Workgroup	1024	256	1024	512	1024	1024	256	256
Vect.Width Float	1	1	4	1	4	4	4	4
Clock (MHz)	1147	1400	850	1404	2933	2600	3200	3200
Max.Alloc. (MB)	671	256	512	220	1024	1024	757	757
Images	Yes	Yes	Yes	Yes	No	No	No	No
Kernel Args	4352	4352	1024	4352	4096	4096	256	256
Alignment	64	128	128	16	128	128	1	1
Cache	R/W	None	R/W	None	R/W	R/W	R/W	None
Cache Line	128	-	128	-	64	64	128	-
Cache Size (KB)	224	-	112	-	64	64	32	-
Global Mem (MB)	3072	1024	2048	877	3072	3072	3072	3072
Constant (KB)	64	64	64	64	64	64	64	64
Local Type	Scratch	Scratch	Scratch	Scratch	Global	Global	Global	Scratch
Local (KB)	48	32	48	16	32	32	512	243
Timer Res. (ns)	1000	1000	1	1000	1	1	37	37

clusters can work on different data using different threads that issue the same instruction, a method referred to as Single Instruction Multiple Thread (SIMT).

AMD Radeon HD5870. The Cypress GPU on this card has 20 compute units containing 16 Very Long Instruction Word (VLIW) [5] processors with an instruction word length of five. To benefit from the VLIW architecture in OpenCL the programmer should use a vector data type such as `float4`.

NVIDIA GeForce GTX460. The GTX460 contains a GF110 Fermi GPU which comprises 7 compute units. These compute units are similar to the ones on the TESLA 2050, with one important difference. Each compute unit consists of 3 SIMT clusters fed by 2 superscalar scheduling units.

NVIDIA GeForce GTX275. This graphics card is based on the GT200 GPU which has 30 compute units containing 8 scalar processors which work in SIMT manner.

Intel Xeon X5570. The Intel Xeon X5570 features 4 physical CPU cores with simultaneous multithreading (SMT) leading to a total of 8 logical cores. The Xeons used in our benchmarks are mounted on an IBM HS22 Blade featuring two CPUs with shared main memory, resulting in a single OpenCL device with a 16 compute units.

AMD Opteron 2435. The Opteron 2435 CPUs used in this paper are mounted on a two-socket IBM LS22 Blade. Each of them contains 6 cores leading to a total of 12 compute units.

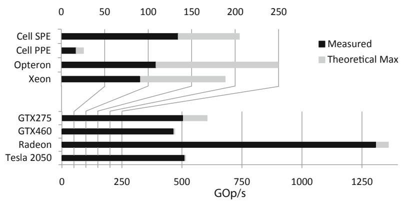
IBM PowerXCell 8i. The accelerator device in our benchmarks consists of two PowerXCell 8i processors mounted on an IBM QS22 Blade. In OpenCL a Cell processor comprises two devices: A CPU (the PPE of the Cell) and an accelerator (all SPEs of the Cell). The two Cell PPEs, each featuring SMT, contain four compute units, the eight SPE cores of the two Cell chips add up to 16 compute units.

3.1 Arithmetic Throughput

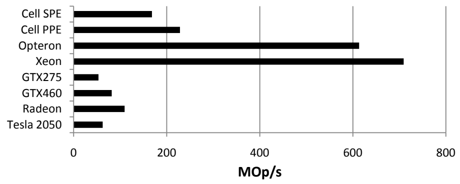
We have gathered well over 3000 throughput measurements using the uCLBench arithmetic benchmark. A small subset that provides an overview of the devices and contains some of the more significant and interesting results will be presented in this section.

Fig. 2(a) shows the number of floating point multiplications per second measured on each device and the theoretical maximum calculated from the hardware specifications. The first thing to note is the large advantage of GPUs in this metric, which necessitates the use of separate scales to portrait all devices meaningfully.

Looking at the effective utilization of hardware capabilities, the GPUs also do well. The Fermi cards reach over 99% utilization. The other GPUs still go



(a) Parallel



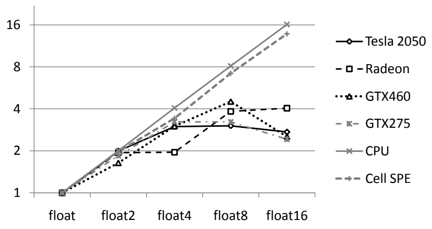
(b) Sequential

Fig. 2. Floating point multiplication throughput

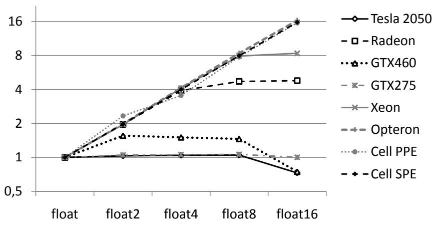
over 80% while the two x86 CPUs fail to reach the 50% mark. IBM's OpenCL performs a bit better, achieving slightly over 65% of the theoretical maximum throughput on both PPEs and SPEs.

While throughput of vectorized independent instructions is important for scientific computing and many multimedia workloads, some problems are hard to parallelize. The performance in such cases depends on the speed at which sequentially dependent calculations can be performed, which is summarized in Fig. 2(b). The CPUs clearly outperform GPUs and accelerators here, providing a solid argument for the use of heterogeneous systems.

Vectorization. Figures 3(a) and 3(b) show the relative performance impact of manual vectorization using the `floatN` OpenCL datatypes. With a single work item all devices benefit from vectorization to some extent. Since all three CPUs deliver the same relative performance, they are consolidated.



(a) Using a single Work Item



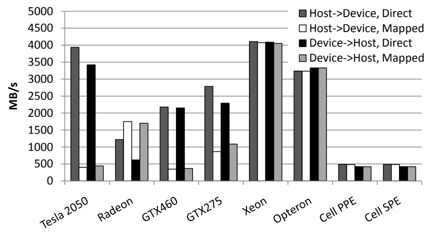
(b) Using many Work Items

Fig. 3. Vectorization Impact

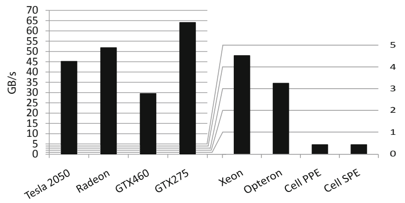
When the full amount of work items is used there are two clearly visible categories. The NVIDIA GPUs effectively gather individual work items into SIMD groups and thus show no additional benefit from manual vectorization, vectors with 16 elements even slow down execution. The GTX460 result is counter-intuitive, but can be explained by scheduling constraints introduced by the superscalar architecture.

3.2 Memory Subsystem

The memory subsystems of the benchmarked OpenCL devices diverge in two areas – availability of dedicated global device memory and structure of the on-chip memory. The GPU devices feature dedicated global memory while for all other devices the global device address space resides in host memory. Furthermore, the local memory on GPUs and Cell SPUs is a manually managed scratchpad while on CPUs it is placed inside the cache hierarchy.



(a) Bandwidth transferring data from/to the device



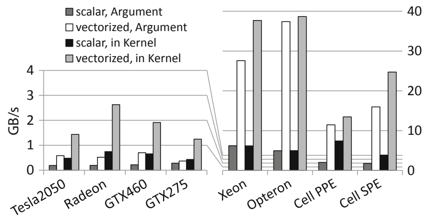
(b) Global device memory bandwidth

Fig. 4. Bandwidth measurements

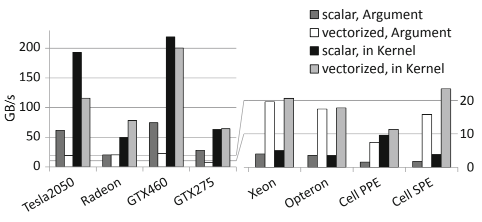
Bandwidth. The bandwidth measured between host and devices is shown in Fig. 4(a). For CPUs data is simply copied within the main memory, while for GPUs it has to be transferred over the PCIe bus. Therefore the bandwidth measured for CPUs is higher in this benchmark,and the results of the two CPUs using the AMD implementation correspond to their main memory bandwidth. All NVIDIA GPUs perform similarly, whereas the Radeon is far behind them using direct memory while it is faster when using mapped memory. The Cell processor achieves very low bandwidth although it is equipped with fast memory, a result that we attribute to an immature implementation of the IBM OpenCL runtime.

A second property we measured is the bandwidth of the devices' global memory. As shown in Fig. 4(b) the GPUs lead the benchmark due to their wide memory interface. The GTX275 outperforms the Radeon as well as the newer NVIDIA GPUs although the theoretical memory bandwidth of the latter ones is slightly higher. All CPUs achieve the same bandwidth as in the host $\leftrightarrow$ device benchmark since host and device memory are physically identical.

Looking further into the memory hierarchy we measure the bandwidth of a single compute unit to its local memory. Since all compute units on a device can access their local memory concurrently, the numbers provided need to be multiplied by the compute unit count to calculate the local memory bandwidth of the whole device. We measured the bandwidth in four ways: in the first case only one work item accesses the memory, in the second the maximum launchable number is used. These two variants were used on local memory that has been statically declared inside a kernel function as well as to local memory passed as



(a) Using a single work item



(b) Using multiple work items

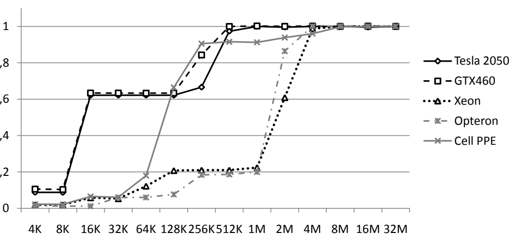
Fig. 5. Bandwidth of one compute unit to its local memory

an argument to the kernel function. Furthermore, all benchmarks were performed using scalars and vector data types. Fig. 5(a) shows the result of the benchmarks using only one work item while Fig. 5(b) displays the values for the full amount. GPU scratchpad memories are clearly designed to be accessed by multiple work items, and with parallel access performance increases by up to two orders of magnitude.

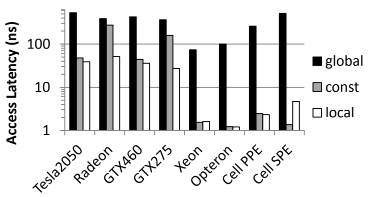
In contrast to the GPUs the Cell SPE scratchpad memory can be used efficiently in the sequential benchmark, parallelizing the access has only a minor impact on the speed. On the CPU side, all systems exhibit unexpected slowdowns with multiple work items. We believe that this is caused by superfluous cache coherency operations due to false sharing [11]. All CPUs benefit from vector types, and all devices can utilize higher bandwidth to the local memory when it is statically declared inside the kernel function.

Latency. One purpose of the multiple address spaces in OpenCL is allowing access to lower latency memory pools. This is particularly important on GPUs and accelerators, where global memory is often uncached. As shown in Fig. 6(b) absolute access latency to global memory is almost an order of magnitude larger on GPUs and accelerators than on CPUs. Additionally CPUs can rely on their highly sophisticated cache hierarchies to reduce the access times even further. The impact of caching is shown in Fig. 6(a) which shows the relative time to access a data item of a certain size in comparison to the previously measured latency to the global memory. This depiction clearly identifies the number of caches featured by a device, as well as their usable size in OpenCL. Non-Fermi GPUs as well as the Cell SPE do not feature any automated caching of data in global memory resulting in equal access time for all tested sizes.

Local and Constant memory latency is significantly smaller on all devices. On the CPUs it corresponds to L1 cache latency as expected. All four GPUs show very similar results to access the local memory, while the Fermi based chips outperform the Radeon and GTX275 in accessing the constant memory by approximately six and three times, respectively. The accelerator's behavior more closely resembles a CPU than a GPU regarding local latency, resulting in the largest difference between global and local timings. The SPEs are the only device to achieve significantly lower latency for `const` than `local` accesses.



(a) Global depending on size



(b) Measured in absolute time

Fig. 6. Memory access latency

3.3 Branching Penalty

We measured the time taken to process the branch penalty testing kernel with one to 128 branches relative to the time required to complete a single branch. All CPUs remain at the same performance level regardless of the number of divergent branches. This is expected, as CPUs do not feature the SIMT execution model that results in a branching penalty. The Cell SPE accelerator also does not exhibit any penalty. The situation is more interesting for the GPUs, which show a linear increase in runtime with the number of branches until a cutoff point. In case of all NVIDIA GPUs, this point is reached at 32 divergent branches, and it takes 64 branches on the Radeon. This measurement coincides perfectly with the *warp size* reported for each GPU, which is the number of SIMT threads that are grouped for SIMD execution.

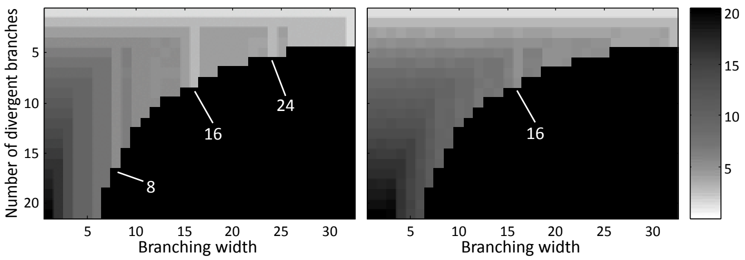


Fig. 7. Branching penalty with varying branch width. NVIDIA GPUs / Radeon

Fig. 7 summarizes the results obtained varying both branch count and topological layout of branches in the local range. A darker color indicates longer kernel runtime, and the lower right part is black since it contains infeasible combinations of branching width and branch count. Generally, grouping branches together improves performance. In fact, the hardware behaves in a very predictable way: if the condition $branchingWidth * branchCount \geq warpSize$ is fulfilled, further increases in the branch count will not cause performance degradation. On NVIDIA GPUs, multiples of 8 for the branch width are particularly advantageous, and the same is true for multiples of 16 on the Radeon. For GTX275 and Radeon this value is equal to the reported SIMD width of the architecture. This is not the case for the Fermi-based NVIDIA GPUs, where a SIMD width of 16 is generally assumed, yet their behavior remains unchanged.

3.4 Runtime Overheads

Invocation overheads remain below 10 microseconds on the tested x86 CPUs as well as the Fermi GPUs. The two IBM Parts and the GTX275 take around 30 and 50 microseconds, respectively. The Radeon HD5870 requires approximately 450 microseconds from enqueueing to kernel startup.

We measured compilation times below 1 second for all mature platforms, scaling linearly with code size. The IBM platform has larger compilation times, particularly for the SPEs, reaching 30 seconds and more for kernels beyond 200 lines of code.

4 Guiding Kernel Design

In this section we evaluate the usefulness and accuracy of the device characterization provided by the uCLbench suite on a real-world kernel. Performance portability is a main concern with OpenCL kernels, with different devices reacting very differently to optimization attempts. While some of these optimizations – such as local work group dimensions – can be auto-tuned relatively easily others require significant manual implementation effort. These latter optimizations are the focus of this chapter. We will demonstrate that the automatic characterization provided by uCLbench reliably identifies promising optimizations for each of the diverse set of devices tested, and that implementing these optimizations consistently improves performance beyond the capabilities of a state-of-the-art optimizing GPGPU compiler.

4.1 The Model Problem

As our test case we selected a simple elliptic partial differential equation, the discrete two-dimensional Poisson equation with Dirichlet boundary conditions in the unit square $\Omega = (0,1)^2$. This problem is given by

$$\begin{aligned} -\Delta_h u_h(x,y) &= f_h^{\Omega}(x,y) \\ u_h(x,y) &= f_h^{\Gamma}(x,y) \quad \text{for} \quad ((x,y) \in \Gamma_h = \partial\Omega_h) \end{aligned}$$

with boundary conditions $f_h^{\Gamma}(x,y)$ and discretization width $h = 1/n$, $n \in I\!N$ being the number of grid points in each direction.

The central component in a multigrid algorithm for this problem is the relaxation step, an iterative solver for the discretized version of the given equation. Due to its good smoothing effects and parallelization properties (see [12]), we chose an ω-Jacobi kernel.

4.2 Optimizations

To evaluate the predictive power of our benchmark suite, we implemented six kinds of manual optimizations belonging to one of three categories:

Vectorization. The kernel was vectorized for vector widths of 4, 8 and 16. (designated vec4, vec8, vec16)

Branching Elimination. A straightforward implementation of the boundary conditions $f_h^{\Gamma}(x, y)$ introduces dynamic branching. This optimization eliminates the branching by using oversized buffers and performing a separate step to fix the boundary after each iterative step. (designated BE)

Manual Caching. These kernels manually load their working set into local memory. There are two slightly different versions, *dynamic* and *static*. The former passes the local memory buffer to the kernel as a parameter while the latter statically allocates it inside the kernel. (designated mcDyn and mcStat)

Combinations of these optimizations result in a total of 16 implementations, plus one baseline version with no manual optimization. This does not include the variation introduced by easily tunable parameters like local work group dimensions – for these, we selected the optimal values for each device by exhaustively iterating over all possibilities. Clearly, manually implementing 16 or more versions of each kernel is not generally viable. The following results demonstrate that automatic characterization can be used to assess the impact of an optimization on a given device before implementing it, thus guiding the implementation effort.

4.3 Results

Table 2 lists, for each device, the best performing version of the ω-Jacobi kernel and how much that version improves upon the baseline (Improvement = $(T_{\text{baseline}}/T_{\text{best}} - 1) * 100$). If the best version combines more than one optimization, the *Primary* column contains the single optimization that has the largest impact on the result, and the *Contribution* column shows the impact of that option on its own. Finally, we present the speedup achieved by a state of the art GPGPU optimizing compiler targeted primarily at the GTX275 architecture [15]. Even when a speedup is achieved, the automatically optimized kernel still does not reach the performance of the version arrived at by guided manual optimization.

The results correspond to several device characteristics identified by our benchmarks. All the devices that do not show any caching behavior in the memory latency tests – Radeon, GTX275 and Cell SPE – benefit from manual caching, and as determined by the local memory bandwidth results static allocation is generally as fast or faster than the alternative (Section 3.2). Branch elimination is most beneficial on those devices that show a high variance in the branch penalty benchmark, while not having any impact on CPUs with their flat branching profile (Section 3.3). In this regard, the minor Cell PPE speedup due to BE seems

Table 2. Most Effective Optimizations per Device and their Speedup

Device	Best Version	Improvement	Primary	Contrib.	GPGPU Compiler
Tesla 2050	vec4	14%	-	-	slowdown
Radeon	vec8_BE_mcStat	63%	vec8	29%	crashed
GTX460	vec4_BE	10%	vec4	9%	slowdown
GTX275	BE_mcStat	31%	mcStat	19%	22%
Xeon	vec8	42%	-	-	slowdown
Opteron	vec16	79%	-	-	slowdown
Cell PPE	vec16_BE	39%	vec16	25%	slowdown
Cell SPE	vec16_BE_mcStat	192%	mcStat	78%	90%

counter-intuitive, but its arithmetic throughput results indicate that it suffers penalties in branching code due to its long pipeline and in-order execution.

The impact of vectorization is well predicted by the characterization results (Section 3.1). While the NVIDIA compiler and devices do a good job at automatically vectorizing scalar code, on all other platforms the impact of manual vectorization is large. Our vectorization benchmark results correctly indicate the most effective vector length for each device.

5 Related Work

Microbenchmarks have a long history in the characterization of parallel architectures. The Intel MPI Benchmarks (IMB) [7] are often used to determine the performance of basic MPI operations on clusters. For OpenMP, the EPCC suite [2] measures the overheads incurred by synchronization, loop scheduling and array operations. Bandwidth is widely measured using STREAM [8], and our memory bandwidth benchmark implementation is based on its principles.

A major benefit of using OpenCL is the ability to target GPU devices. Historically these were mostly used for graphics rendering, and benchmarked accordingly, particularly for use in games. A popular tool for this purpose is 3DMark [10]. When GPU computing first became widespread Stanford University's GPUbench suite [1] provided valuable low-level information. However, it predates the introduction of specific GPU computing languages and platforms, and therefore only measures performance using the restrictive graphics programming interface. In depth performance analysis of one particular GPU architecture has been performed by Wong et al. [14].

Recently the SHOC suite of benchmarks for OpenCL was introduced [4]. While it contains some microbenchmarks, it is primarily targeted at measuring mid- to high-level performance. It does not try to identify the individual characteristics of mathematical operations or measure the latency of access to OpenCL address spaces. Conversely, our suite is aimed at determining useful low-level characteristics of devices and includes exhaustive latency and arithmetic performance measurements as well as a benchmark investigating dynamic branching penalties. We also present results for a broader range of hardware, including an accelerator device.

The Rodinia Heterogeneous Benchmark Suite [3] predates wide availability of OpenCL, therefore separately covering CUDA, OpenMP and other languages with distinct benchmark codes. Also, unlike uCLbench, Rodinia focuses on determining the performance of high-level patterns of parallelism.

6 Conclusion

The uCLbench suite provides tools to accurately measure important low-level device properties including: arithmetic throughput for parallel and sequential code, memory bandwidth and latency to several OpenCL address spaces, compilation time, kernel invocation overheads and divergent dynamic branching penalties. We obtained results on eight compute devices which reflect important hardware characteristics of the platforms and, in some cases, show potential for improvement in the OpenCL implementations.

The automatic device characterization provided by uCLbench is useful in quickly gaining an in-depth understanding of new hardware and software OpenCL platforms, exposing undisclosed microarchitectural details such as dynamic branching penalties. We have shown that the measured characterisics can be used to guide manual optimization by identifying the most promising optimizations for a given device. Applying these transformations to an ω-Jacobi multigrid relaxation kernel results in an average improvement of 61% across the devices tested.

Acknowledgments. This research is partially funded by the Austrian Research Promotion Agency (FFG), project number: 824925. We thank Dr.techn. Peter Zinterhof from the University of Salzburg for giving us access to a Tesla cluster.

References

[1] Buck, I., Fatahalian, K., Hanrahan, P.: GPUBench (2004)

[2] Bull, J.M.: Measuring synchronisation and scheduling overheads in openmp. In: Proc. of 1st Europ. Workshop on OpenMP, pp. 99–105 (1999)

[3] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: IEEE Workload Characterization Symposium, pp. 44–54 (2009)

[4] Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S.: The scalable heterogeneous computing (shoc) benchmark suite. In: GPGPU 2010: Proc., pp. 63–74. ACM, New York (2010)

[5] Fisher, J.A.: Very long instruction word architectures and the ELI-512. In: Proceedings of the 10th Annual International Symposium on Computer Architecture, pp. 140–150. ACM, New York (1983)

[6] Fung, W.L., Sham, I., Yuan, G., Aamodt, T.M.: Dynamic warp formation and scheduling for efficient gpu control flow. In: MICRO 40, pp. 407–420. IEEE Computer Society, Washington, DC, USA (2007)

[7] MPI Intel. Benchmarks: Users Guide and Methodology Description. Intel GmbH, Germany (2004)

[8] McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. IEEE Comp. Soc. Tech. Comm. on Computer Architecture (TCCA) Newsletter, pp. 19–25 (December 1995)

[9] Olukotun, K., Hammond, L.: The future of microprocessors. Queue 3(7), 26–29 (2005)

[10] Sibai, F.N.: Performance analysis and workload characterization of the 3dmark05 benchmark on modern parallel computer platforms. ACM SIGARCH Computer Architecture News 35(3), 44–52 (2007)

[11] Torrellas, J., Lam, M.S., Hennessy, J.L.: False sharing and spatial locality in multiprocessor caches. IEEE Transactions on Computers 43(6), 651–663 (1994)

[12] Trottenberg, U., Oosterlee, C.W., Schueller, A.: Multigrid. Academic Press, London (2001)

[13] Volkov, V., Demmel, J.W.: Benchmarking gpus to tune dense linear algebra. In: SC 2008, pp. 1–11. IEEE Press, Piscataway (2008)

[14] Wong, H., Papadopoulou, M., Sadooghi-Alvandi, M., Moshovos, A.: Demystifying gpu microarchitecture through microbenchmarking. In: ISPASS, pp. 235–246 (2010)

[15] Yang, Y., Xiang, P., Kong, J., Zhou, H.: A gpgpu compiler for memory optimization and parallelism management. In: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation, PLDI 2010, pp. 86–97. ACM, New York (2010)