The Future of Accelerator Programming: Abstraction, Performance or Can We Have Both?

Kamil Rocki[*†]
kmrocki@us.ibm.com

[*]IBM Research
650 Harry Road
San Jose, CA 95120
United States

Martin Burtscher[‡]
burtscher@txstate.edu

[†]University of Tokyo
Department of Computer Science
7-3-1 Hongo, Bunkyo-ku
Tokyo, Japan

Reiji Suda[†]
reiji@is.s.u-tokyo.ac.jp

[‡]Texas State University
Department of Computer Science
San Marcos, TX 78666
United States

ABSTRACT

In a perfect world, code would only be written once and would run on different devices with high efficiency. To a degree, that used to be the case in the era of frequency scaling on a single core. However, due to power limitations, parallel programming has become necessary to obtain performance gains. But parallel architectures differ substantially from each other, often require specialized knowledge to exploit them, and typically necessitate reimplementation and fine tuning of programs. These slow tasks frequently result in situations where most of the time is spent reimplementing old rather than writing new code.

The goal of our research is to find programming techniques that increase productivity, maintain high performance, and provide abstraction to free the programmer from these unnecessary and time-consuming tasks. However, such techniques usually come at the cost of substantial performance degradation. This paper investigates current approaches to portable accelerator programming, seeking to answer whether they make it possible to combine high efficiency with sufficient algorithm abstraction. It discusses OpenCL as a potential solution and presents three approaches of writing portable code: GPU-centric, CPU-centric, and combined. By applying the three approaches to a real-world program, we show that it is at least sometimes possible to run exactly the same code on many different devices with minimal performance degradation using parameterization. The main contributions of this paper are an extensive review of the current state-of-the-art and our original approach of addressing the stated problem with the *copious-parallelism* technique.

Categories and Subject Descriptors

C.1.2 [**Computer Systems Organization**]: PROCESSOR ARCHITECTURES—*Single-instruction-stream, multiple-data-stream processors (SIMD)*

SAC'14 March 24-28, 2014, Gyeongju, Korea.

General Terms

Algorithms, Performance, Design

1. INTRODUCTION

Parallel computing has become a necessity as it provides increased computational performance while reducing the energy consumption. Hence, the general hardware trend is towards more parallelism in form of additional cores and wider vectors (e.g., SSE and AVX instructions). To keep up with these developments, software needs to be equally parallelized. In fact, the advent of multi-core CPUs meant, for the first time, that existing serial code would only run faster on new hardware if it was rewritten in a multithreaded way.

The single most important feature of multithreading is that, if the number of threads is sufficiently high, an arbitrary number of processing elements can be utilized. Running such a program on an older CPU with fewer cores typically still results in good performance for that CPU. Hence, it is reasonable to assume that this technique, i.e., providing enough parallelism, can result in portable high-performance code. This leads to the now obvious conclusion that if code written in the year 2000 had been designed in a sufficiently parallel manner, we would observe speedups to this day.

We are now facing a similar albeit even more complex problem with Accelerated Processing Units (APUs) such as FPGAs, GPUs, and MICs. The architectures of these devices differ drastically and greatly affect how a program has to be written to achieve good performance. Even within a device family, successive generations often differ substantially, making it unclear how to implement programs that run efficiently on future accelerators. Yet, one would like to be able to simply execute previously written programs on newer accelerators and obtain speedups that are proportional to the hardware development. This would allow programmers to focus on improving the algorithms rather than having to reimplement them every time a new version of an accelerator is released.

As CPUs and APUs are likely to be merged in the future, simple multithreading will no longer suffice and parallelism will have to be exploited in a different way. By definition, APUs accelerate computations outside of the CPU, thus creating the first obstacle, which is CPU-APU communication and memory addressing. Currently, CPU/GPU or multi-GPU execution has to be expressed explicitly. This is especially worrisome since it 'stretches' the concept of uniform programming quite a bit. Another problem is efficient load

balancing as every device in a heterogeneous system can have different throughput and latency limitations. If future PCs are similar to current supercomputers, they may contain entire clusters of physically or virtually separated devices, thus requiring inter-process communication such as MPI.

Future compute devices may have millions of processing elements in tens of thousands of compute nodes spread across hundreds of heterogeneous devices. Clearly, such computers will make the already difficult task of writing parallel programs only more complicated. This makes it absolutely critical to enable programmers to write code once and run it on different devices with reasonable efficiency, that is, the performance on newer hardware should be proportional to the provided compute power without the need for tuning or reimplementing the code. The goal of our work is to help make this a reality.

The main contributions of this paper are as follows. (1) We present an extensive review of the related state-of-the-art research, i.e., of different ways to provide both performance and portability. (2) We describe a novel approach of addressing this problem using OpenCL, which we call the *copious-parallelism* technique. (3) We evaluate our method on a real application and four distinct devices that require very different programming models.

The rest of this paper is organized as follows. Section 2 surveys related work. Sections 3 and 4 present our approach. Section 5 describes our test application as well as different ways to parallelize it. Section 6 analyzes the performance results. Section 7 concludes our study.

2. RELATED WORK

2.1 Halide

Halide demonstrates that both performance and abstraction can be achieved within a domain by separating the algorithm description, which defines the storage, from the schedule, which defines the order of computation tasks [1]. This separation allows the same program to behave differently on different devices simply by altering the execution schedule. When recompiling code for each platform, the performance achieved on architectures ranging from ARM-based CPUs to GPUs is the same as that of hand-tuned assembly utilizing SIMD instructions. However, there is a caveat. The domain is restricted to image processing, a task that is very parallel in nature. This limitation allows tuning the compiler appropriately. Thus, Halide combines a very impressive compiler for a specific domain with an elegant way to make code portable and modular. This approach raises interesting questions. Can abstraction and performance only be achieved together at the cost of domain restriction? How much a priori knowledge about the underlying algorithms and data structures is required? What would be the cost of implementing such a compiler for other domains?

2.2 Phalanx

Phalanx is an architecture-aware C++-like programming language [2]. It addresses large-scale parallelism, heterogeneous devices, and complicated memory hierarchies. The background of this project is the Echelon processor concept (NVIDIA's Extreme-Scale Computing Project) whose architecture is similar to current supercomputer nodes comprising both GPU-like throughput and CPU-like latency-optimized cores. This design reduces energy consumption by putting both types of cores onto one chip and by reducing data movement costs. The project might indicate that future commodity hardware will be similar to current supercomputers whose programming can be quite difficult. Hence, the main goal of Phalanx is to simplify the programming by (1) creating a unified model for heterogeneous parallel machines, i.e., a single notation that captures all processor types, and (2) designing it in such a way that it likely works for current and future machines.

Phalanx is able to obtain information about the underlying hardware and memory hierarchy and execute kernels accordingly. The backend supports CUDA for running on NVIDIA GPUs, OpenMP for CPUs, and GASNet for multi-node execution. The preliminary results are very promising. Algorithms such as MatMul and FFT scale extremely well on clusters based on CPUs (Cray XE6, IBM Blue Gene/P) and GPUs (Cray XK6). This concept provides many features that we would expect from a programming language for future accelerators. However, the work is still quite preliminary and the CUDA backend restricts the supported accelerators to NVIDIA GPUs.

2.3 C++ AMP

C++ Accelerated Massive Parallelism (C++ AMP) is a library implemented on DirectX 11 and an open specification from Microsoft for implementing data parallelism directly in C++ [3]. The C++ AMP programming model includes multidimensional arrays, indexing, memory transfer, tiling, and a mathematical function library. The most powerful feature of C++ AMP is hiding all low-level code from the programmer. Based heavily on C++ and abstraction, an accelerator is represented by a single class and the high-level code remains the same regardless of the used hardware. Vendors are responsible for providing proper implementations. *Parallel for each* statements encapsulate parallel code. No more than a few lines of code are needed to execute a simple parallel code section, making it similar to Halide but domain independent. The schedule is specified as parameters of the *parallel for each* statement (tiling). In contrast to Halide, AMP's performance is far from perfect, suffering particularly from high latencies. Our experiments have shown that, although very high performance can be reached on devices such as Radeon and NVIDIA GPUs, substantial time is needed to run a small task, making AMP impractical for accelerating small problems. Another drawback is being tied to Windows and Visual Studio 2012, which makes the code platform dependent, and the early development stage, which causes problems such as a lack of CPU support.

2.4 OpenCL

The Open Computing Language (OpenCL) is a well-known parallel programming framework for heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other types of processors [4]. It is supported by many operating systems and based on C99, making it very portable. Once written in a parallel manner, OpenCL code can be executed on a variety of devices. The underlying implementation of OpenCL calls relies on so called platforms provided by hardware vendors like AMD, NVIDIA, Intel, and IBM and is therefore hidden from the programmer. The way the code is compiled and executed on a specific device is also handled by the manu-

facturer's implementation.

OpenCL organizes the programs into workgroups that run threads. Similar to CUDA, the threads are very lightweight. Whereas recent research has shown that tweaked OpenCL code performs no worse than CUDA code [5], our results show a performance gap of roughly 20% mainly due to loop unrolling limitations. A big advantage of OpenCL is its ability to run on CPUs. We have observed speedups on CPUs from multithreaded execution as well as from auto-vectorization using Intel's OpenCL platform. OpenCL determines what device to select for execution at runtime and compiles the code when needed (or offline). Similar to CUDA, OpenCL offers full control over the execution, which is why many calls are low-level in nature, making the code design cumbersome at times. Being an open and free standard makes it very portable and widely available. The recently released Xeon Phi clearly demonstrates that new devices do and probably will support OpenCL, allowing programmers to keep the same code and enjoy speedups, which is a perfect example of portability.

2.5 Shevlin Park

Intel is working on a project called Shevlin Park. It augments CLANG and LLVM [6] with C++ AMP and uses OpenCL as its underlying compute backend. The goal of this project is to combine C++ AMP's elegant code design and simplicity with OpenCL's portability and performance. Shevlin Park accomplishes this by translating C++ AMP source code into OpenCL using the CLANG front end and the LLVM back end. As a result, programs can be implemented in a very abstract way using C++ AMP, yet the resulting code will be portable OpenCL. Preliminary performance comparisons show a significant improvement over C++ AMP, but slightly (around 10% on average) worse efficiency compared to native OpenCL. This initiative exploits code translation to achieve good performance while maintaining abstraction and providing just a minimal description of the algorithm as in C++ AMP. Note that Shevlin Park is a concept project and that Intel has not announced any release plans for this technology yet.

2.6 OmpSs

OmpSs is another interesting concept that is close to our goal [8]. It extends OpenMP with new directives to support asynchronous parallelism and heterogeneity. The objective is to keep the same sequential code on any platform. It is based on pragma directives as OpenMP. However, it can also be understood as new directives extending other accelerator-based APIs like CUDA or OpenCL. Whereas the project promises high productivity, the performance bottlenecks and exact benchmarks are not yet reported. Another limitation is that the GPU component currently only supports CUDA.

2.7 Pattern Language

The Pattern Language being developed at UC Berkeley [7] aims at identifying computational patterns and structural patterns [9]. Computational patterns include problems such as N-Body, Dense Matrix Algebra, and Dynamic Programming. Structural patterns include approaches such as MapReduce and Pipe-and-Filter. Identifying those patterns and learning how to solve each of them efficiently leads to a modular design, possibly resulting in specialized libraries or methods that can be quickly applied to a given problem. This approach is different from the ones described above as it assumes that a single programming language is not able to perform well. The main idea is that software architecture is key to the design of parallel programs, and the key to efficient software implementation is frameworks.

3. PROPOSED APPROACH

The idea of *code once, run everywhere* is well known and, as outlined in the previous section, there are many approaches to the problem. The key question is what a programmer needs or what kind of tool maximally increases productivity. This, of course, assumes that three conditions can be satisfied at the same time: the algorithm implementation is abstract, code is always fast (though not necessarily running close to peak performance), and the domain is not restricted. In our opinion, OpenCL currently represents the best basis for such an approach. The low-level implementation details are handled by the vendor and hidden from the user. Thus, it is easy to achieve the first of the stated goals, i.e., code portability on a variety of devices. After all, OpenCL was designed to provide a way of running the same code on different devices. However, being able to execute the code and actually having portable high-performance code are very different things. This is similar to how code written in C can be compiled almost anywhere, but careful machine-dependent optimizations and changes are required to obtain highly efficient implementations. We are assuming, based on the above survey, that programmers are more likely to put effort into additional parallelization and maintain only one version of the code than reimplementing a program for every new device that is released.

Therefore, in order to achieve good performance on many types of devices, we need to exploit parallelism at all levels. For example, our experiments show that, by writing code that is explicitly vectorized (works on vector data types in OpenCL) and that uses a sufficiently large number of threads, peak performance can be approached on CPUs and Intel Xeon Phis as well as NVIDIA and AMD GPUs running exactly the same code. Since this approach seems very simple, one might conclude that it is not a substantial improvement over existing techniques. Yet, none of the aforementioned works rely on a similar method. In addition to being straightforward, our technique proves to be portable and highly efficient. We demonstrate these benefits on a piece of code that we implemented using our technique and tuned for a specific type of device before testing it without changes on hypothetically *new* and *different* hardware. Due to its simplicity, it should be relatively easy to apply our technique to other problems as well.

4. REDUNDANT/EXPLICIT CODING

Our approach is based on the concept of copious parallel programming. Following the same principle as during the shift towards multicore CPUs, we explicitly express the parallel code at the thread and register (vector) level. Additionally, we add explicit local memory management, as is common in GPU programming. Currently, CPUs are not capable of taking advantage of running thousands of threads. However, this might change in the future. Similarly, vector instructions are not (explicitly) used in CUDA code. In the following section, we show how to write code in the proposed copious parallelism style so that it will likely run efficiently

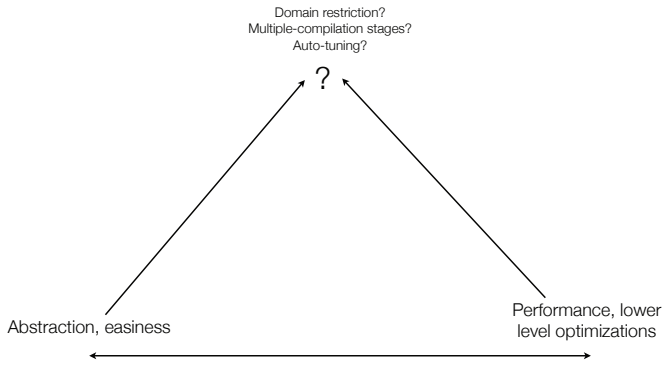

Figure 1: Abstraction vs. performance

on future hardware. We include several intermediate steps to better illustrate how this is done.

4.1 Runtime/compile-time adjustment

The key aspect of our approach is to let the device ignore any redundant parallelism that it does not support. To enable that, the code should be parameterized to allow changes at compile time (using definitions and macros) or at runtime (using variables). This approach mitigates the overhead of the copious parallelism on devices such as CPUs, decreases register usage on GPUs, and increases the flexibility with regard to future hardware. This includes the:

- Number of processors/units
- Number of cores per processor
- Global memory (accessible by all processors)
- Local memory (accessible by one processor or thread)
- Cache (size and hierarchy)
- Clock frequency

4.2 Parallelization

We refer to parallelization as a set of operations being executed simultaneously in an independent manner by more than one thread (a.k.a. Multiple Instruction Multiple Data (MIMD) execution). For example, pthreads and OpenMP can easily express this kind of parallelism.

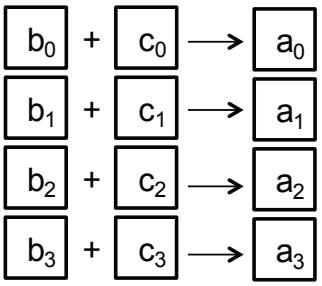

Figure 2: Parallel addition - 4 elements at once, 4 instructions in parallel

4.3 Vectorization

We refer to vectorization as one instruction performing the same operation on more than one element at the same time (a.k.a. Single Instruction Multiple Data (SIMD) execution). For example, SSE registers and instructions allow these kinds of operations (a 128-bit register can hold up to 4 single-precision values).

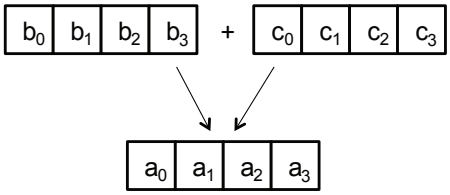

Figure 3: Vector addition - 4 elements at once, 1 instruction

4.4 Explicit memory management

Aside from the number of cores, the main difference between CPUs and GPUs is the distinct approach to local memory management. The off-chip memory (DRAM) has a high latency compared to cache memory. Therefore, to fully utilize GPUs, it is often necessary to perform explicit loads into fast (so-called shared) memory and reuse the stored data. This is similar to how CPU caches operate except the shared memory is controlled by software. Modern GPUs do have caches, but they are not as big as those found on CPUs. CPUs also employ mechanism such as prefetching to preload data into the caches. It is important to exploit the memory hierarchy, and we believe that the techniques developed to program GPUs are also suitable for current and future CPUs as well as other accelerators. As presented in Figure 4, data can be accessed through the cache (typical for CPUs), directly from the memory, or through fast local on-chip memory (typical for GPUs). The usual size of the fast memory (L1 data cache or shared GPU memory is 16-64 kB). An obvious disadvantage of this concept is the necessity to decompose the problem, which might not always be possible.

5. ANALYZED PROBLEM

To evaluate our technique, we chose the well-known n-body problem. This problem has been studied throughout the history of computing [12]. It involves predicting the motion of celestial objects under mutual gravitation. Solving it has been motivated by the desire to understand the motion of the sun, planets, and visible stars. With many objects, n-body simulation can be computationally taxing due to the n^2 force calculations. The parallelism of the n-body problem has also been studied since the advent of parallel computers.

We selected this problem as a test case for several reasons. Due to its relative simplicity, it is easy to identify the degrees of parallelism. Moreover, its results are reproducible, it exhibits significant computational complexity, and the problem size can trivially be adjusted. For these reasons, it is widely used as a benchmark and is, for example, included in the CUDA SDK. The performance is typically expressed in number of interactions calculated per unit time.

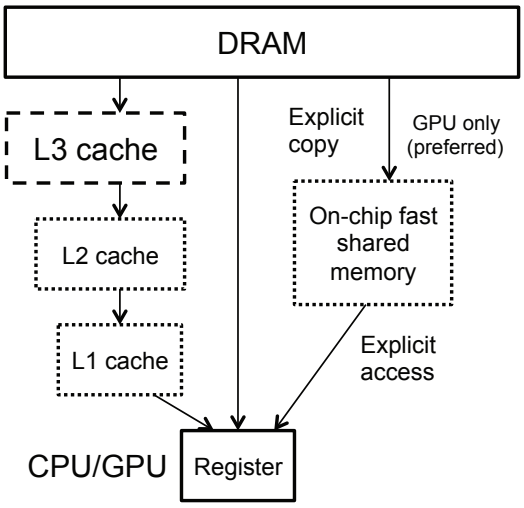

Figure 4: Data access - CPU vs. GPU preferred way

A direct $O(n^2)$ force calculation algorithm computes the interaction of each object with every other object in the system. An interaction comprises roughly 20 to 30 floating-point operations (FLOPs), depending on the implementation. The performance can be presented in FLOP/s (FLOPs per second). Typically, performance grows with increasing problems size (number of bodies). However, due to finite cache sizes, CPUs do not handle large instances well. A method called *cache blocking* is used to solve this problem. Our approach employs a generalization of this technique.

There are additional reasons for our choice that are worth pointing out. Although being a simple problem, implementing *nbody* is not as trivial as it may seem. In particular, recent memory bandwidth/latency limited hardware presents a considerable obstacle. Writing code that performs at 50% of the possible peak performance is very hard. Another reason is that *nbody* is simple enough to illustrate our methodology without requiring a large number of statements. Finally, this choice makes it easy to compare our results to other implementations.

5.1 All-pairs n-body simulation

The total force F_i on body i, due to its interactions with the other $n-1$ bodies, is the sum of all interactions:

$$F_i = \sum_j F_{ij} = G * m_i * \sum_j \frac{m_j * r(i,j)}{r(i,j)^3} \qquad (1)$$

where $r(i,j)$ is the distance between bodies i and j, and m_k is the mass of body k. To integrate over time, the acceleration $a_i = \frac{F_i}{m_i}$ needs to be computed to update the position and velocity of body i, which can be simplified to the following formula [14, 13]:

$$a_i = G * \sum_j \frac{m_j * r(i,j)}{(r(i,j)^2 + \epsilon^2)^{\frac{3}{2}}} \qquad (2)$$

where ϵ is a constant (a softening parameter to improve the accuracy in the presence of discrete time steps).

5.2 Original serial code

The serial code primarily consists of an outer and an inner loop to iterate over the interactions. Then, it updates the acceleration $acc(i)$ and velocity $vel(i)$ based on the calculated force $F(i)$.

for $i \leftarrow 1, n$ **do**
  $A \leftarrow position(i)$ ▷ Read position of i (x,y,z)
  for $j \leftarrow 1, n$ **do**
    $B \leftarrow position(j)$ ▷ Read position of j (x,y,z)
    $M \leftarrow mass(j)$ ▷ Read mass of j
    $F(i) \leftarrow F(i) + Force(A, B, M)$ ▷ Equation (2)
  end for
  Update $acc(i)$, $vel(i)$
end for

$Force(A, B, M)$ comprises 19 floating-point instructions in our implementation (6 additions, 6 multiplications, 6 subtractions, and 1 reciprocal square root; some devices support FMA (Fused-Multiply-Add) instructions that are able to execute the code with as little as 13 floating-point instructions). This implies a high memory throughput and necessitates using cache or shared memory to utilize the arithmetic units efficiently.

5.3 CPU-centric implementation - Inner-loop Vectorization

Perhaps the most straight-forward approach to parallelize the presented code is to divide the outer loop among the available threads and to execute the operations in the inner loop using vector registers and instructions (vector load, vector store, vector add, vector multiply, and vector square root). The inner loop's increment changes according to the vector width. Here, N elements are being processed at the same time (including loads and stores). This also requires the use of vector registers in the outer loop. However, an entire vector can easily be initialized with a single value.

for $i \leftarrow 1, n$ **do** ▷ In Parallel ▷ N = Vector Length
  $A[1, N] \leftarrow position(i)$ ▷ Read 1 element, set N
  for $j \leftarrow 1, n$ **do**
    $B[1, N] \leftarrow position([j, j+N])$ ▷ Read N elements
    $M[1, N] \leftarrow mass([j, j+N])$ ▷ Read N elements
    $F(i) \leftarrow F(i) + Force(A, B, M)$ ▷ N Operations
  end for
  Update $acc(i)$, $vel(i)$
end for

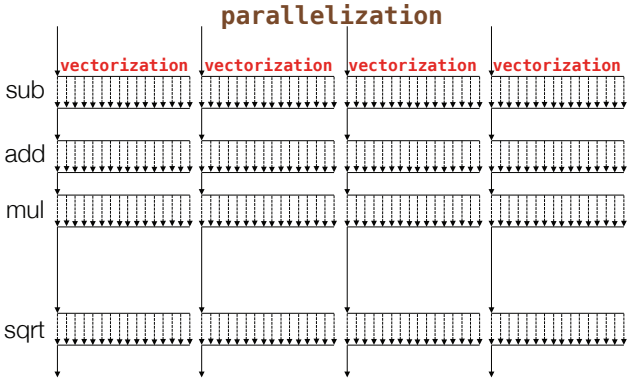

Figure 5: CPU-centric implementation

5.4 CPU-centric implementation - Outer-loop Vectorization

The alternate, less obvious way is to vectorize the loads in the outer loop. This approach minimizes the memory turnaround as the vector loads are performed only N times instead of N^2 times when vectorizing the inner loop. However, there are still $4 * N^2$ single memory accesses in the inner loop to set the values of the vector the registers.

for $i \leftarrow 1, n$ **do** ▷ In Parallel ▷ N = Vector Length
  $A[1, N] \leftarrow position([j, j + N])$ ▷ Read N elements
  for $j \leftarrow 1, n$ **do**
    $B[1, N] \leftarrow position(j)$ ▷ Read 1 element, set N
    $M[1, N] \leftarrow mass(j)$ ▷ Read 1 element, set N
    $F(i) \leftarrow F(i) + Force(A, B, M)$ ▷ N Operations
  end for
  Update $acc(i)$, $vel(i)$
end for

5.5 GPU-centric implementation

A well-known GPU implementations of the $O(n^2)$ n-body algorithm was presented by Nyland et al. [13]. Their implementation completely omits the outer loop and instead uses the thread index as the i value. An important change is the blocking of the inner loop, which preloads chunks of data from the off-chip memory into the on-chip shared memory, enabling the calculations in the inner loop to quickly access and reuse the preloaded data. The block size depends on the size of the shared memory.

$i \leftarrow$ Thread ID
$A \leftarrow position(i)$ ▷ Read position of i (x,y,z)
for $j \leftarrow 1, n$ **do** ▷ Increment by BlockSize
  Read [j, j+BlockSize] into local memory
  for $k \leftarrow 1, BlockSize$ **do**
    Read from local memory
    $B \leftarrow position(k + j)$ ▷ Read position of $k + j$
    $M \leftarrow mass(k + j)$ ▷ Read mass of $k + j$
    $F(i) \leftarrow F(i) + Force(A, B, M)$
  end for
  Update $acc(i)$, $vel(i)$
end for

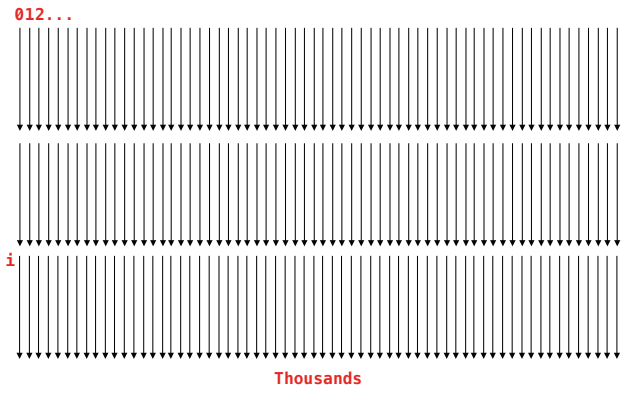

Figure 6: GPU-centric implementation

5.6 Combined approach - Redundant parallelism and explicit memory management

This method combines the outer-loop vectorization with GPU-like blocking to preload the data into fast memory. Each thread again runs one iteration of the outer loop.

$i \leftarrow$ Thread ID ▷ N = Vector Length
$A[1, N] \leftarrow position([j, j + N])$ ▷ Read N elements
for $j \leftarrow 1, n$ **do** ▷ Increment by BlockSize
  Read [j, j+BlockSize] into local memory
  for $k \leftarrow 1, BlockSize$ **do**
    Read from local memory
    $B[1, N] \leftarrow position(k + j)$
    $M[1, N] \leftarrow mass(k + j)$
    $F(i) \leftarrow F(i) + Force(A, B, M)$ ▷ N Operations
  end for
  Update $acc(i)$, $vel(i)$
end for

5.7 Constraints and possible future obstacles

We identified several problems related to our approach that are caused by current limitations of OpenCL. The native CUDA, AVX, and Xeon Phi implementations provide slightly better performance than their OpenCL counterparts. For instance, CUDA allows better control over the register usage and enables control of the L1 cache/shared memory size. In addition, it is able to perform better loop unrolling. Moreover, not all OpenCL drivers translate instructions into machine code in the same way. Specifically, we found that operations such as square root or reciprocal are not always converted in the desired manner. Another limitation of OpenCL when compiling CPU code is the lack of prefetching. Since memory hierarchies will only become more complex, it is important to exploit them well.

Even for our simple program, a large number of parameters describing the cache hierarchy and memory management schemes, which are the primary sources of difficulty in performance portable programming, must be configured. However, as mentioned, we believe programmers are likely to put effort into additional parallelization if it means that they only have to maintain one version of the code. Parametrization is therefore a key aspect of our method.

6. RESULTS AND ANALYSIS

We tested our copious-parallelism technique by comparing fast hardware-optimized implementations of the n-body algorithm to our general OpenCL version that is capable of running on all tested devices. We used four disparate devices for this experiment: an NVIDIA GeForce Titan GPU (CUDA/OpenCL), an AMD Radeon 7970 GPU (OpenCL), an Intel Xeon E5-2690 CPU (Intel Compiler and OpenCL), and an Intel Xeon Phi 5110P MIC accelerator (Intel Compiler and OpenCL). Table 1 lists pertinent information about each tested device. The hardware-specific implementations are carefully tuned for each device. We use their performance as reference results, which are shown in Table 2. The NVIDIA implementation uses CUDA whereas the AMD GPU runs a straight OpenCL port of this code. We tested both Intel devices using implicit (the compiler is capable of vectorizing the code to some extent automatically) and explicit vectorization. Table 2 shows that our hardware-specific implementations achieve over 80% of the possible peak performance. The codes execute 7 normal floating-

point operations and 6 Fused-Multiply-Adds (FMADs), for a total of 19 FLOPs per interaction. Given the ability of certain devices to run an FMAD instruction in every cycle, we can only achieve approximately 73% of the theoretical peak performance (which assumes executing FMADs exclusively) of the Intel Xeon Phi, NVIDIA GTX Titan, and AMD Radeon. Just like the GPU-based OpenCL implementation is a translation of the optimized CUDA code, the CPU-based OpenCL implementation is a port of the CPU native implementation.

Table 3 presents results for the GPU-based OpenCL implementation running on each device. The table shows the performance in FLOP/s as we want a measure of relative efficiency (runtime does not reflect how well a program performs compared to the maximum possible speed). Operations not related to the kernel execution have negligible impact on the runtime. Whereas the code performs well on both GPUs, it does not on the Intel devices. Apparently, it can be efficiently parallelized by substituting the outer loop with thread indexing even without vectorization. Table 4 shows the results of translating the CPU code with inner-loop vectorization to OpenCL.

Outer-loop vectorization combined with local memory blocking appears to be the best solution, running well on all tested devices, as highlighted in Table 5. Surprisingly, the performance of this implementation on the Radeon GPU is even better than using the code ported from CUDA. One GPU limitation appears to be the increased register usage when using explicit vectorization (such as float16, float8, or float4 data types). This causes the compiler to reserve more resources and the GPU is not able to run as many threads in parallel. Therefore, code parameterization is important in this case to change the vector size. We have also noticed that the OpenCL version is inferior compared to the native CPU code when fast approximate instruction such as 23-bit reciprocal vector square root can be used (Table 5). OpenCL does not seem to be capable of utilizing such instructions.

7. CONCLUSIONS

This paper investigates current approaches to portable accelerator programming, seeking to answer whether it is possible to combine high efficiency with algorithm abstraction or not. Our main contributions are an extensive review of the current state-of-the-art and our original approach of addressing the issue with a generalized copious-parallelism technique using OpenCL. Parallel architectures can differ substantially from one another and often require specialized knowledge to exploit them. Sometimes, complete reimplementation and fine tuning of code is necessary for a specific device. This often leads to a situation where most of the time is spent reimplementing old rather than writing new code. Whereas the OpenCL language provides portability, its efficiency still heavily depends on the implementation.

This paper shows three approaches of writing portable OpenCL code: GPU-centric, CPU-centric, and combined. All three approaches have been tested on a real application. The results indicate that it is possible to run exactly the same code on many different devices with minimal performance degradation. We present a high-performance implementation of the classic n-body algorithm using CUDA on a NVIDIA GPU, OpenCL on AMD GPU as well as native Intel instructions on Xeon and Xeon Phi devices. We show the intermediate steps of obtaining a portable implementation using our copious parallel programming technique and describe how to write such code so that it will likely run efficiently on future hardware. We are assuming (based on the survey presented earlier) that programmers are more likely to put effort into additional parallelization and maintain only one version of their code rather than reimplement a program for every new device. We believe there is a group of invariants, such as the memory hierarchy, the number of cores, and the clock frequency, whose values can be parametrized. Therefore, assuming that future devices do not fundamentally change, code written using our technique should remain portable.

Since our method seems simple, one might conclude that it is not a substantial improvement over the existing techniques. However, Halide, Phalanx, C++ AMP, Shelvin Park, and OmpSs do not use or rely on a similar approach. Moreover, our method proves to be very effective, meaning it is portable and highly efficiently, in addition to being relatively simple. In all cases, we are able to achieve over 80-90 percent of peak performance for the given mix of floating-point operations using native implementation and over 75 percent with our portable OpenCL source code. This shows that it is possible to achieve high performance with portable code.

Our future plans include testing a wider set of problems, including irregular applications and data-intensive codes. We would also like to test our proposed method on more devices supporting OpenCL such as FPGAs and DSPs.

Acknowledgments

This work was supported by the Core Research of Evolutional Science and Technology (CREST) project of the Japan Science and Technology Agency (JST), a Grant-in-Aid for Scientific Research of MEXT Japan, and the US National Science Foundation grants 1141022 and 1217231.

8. REFERENCES

[1] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, F. Durand. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. ACM Transactions on Graphics (TOG) - SIGGRAPH 2012 Conference Proceedings. Volume 31 Issue 4, July 2012

[2] M. Garland, M. Kudlur, Y. Zheng. Designing a Unified Programming Model for Heterogeneous Machines. SC '12 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis

[3] C++ AMP Overview `http://msdn.microsoft.com/en-us/library/vstudio/hh265137.aspx`

[4] OpenCL - The open standard for parallel programming of heterogeneous systems `http://www.khronos.org/opencl/`

[5] J. Fang; Varbanescu, A.L.; Sips, H. A Comprehensive Performance Comparison of CUDA and OpenCL. Parallel Processing (ICPP), Conference, 2011

[6] clang: a C language family frontend for LLVM `http://clang.llvm.org/`

[7] K. Asanovic, R. Bodik, B. Christopher Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. University of California, Berkeley,

Device (Type)	Compute Units	SIMD	Clock	FMAD	Peak FLOP/s (32-bit)
NVIDIA GTX Titan	14 x 192 Cores	per 32 Threads	837 MHz	Yes	4.5 TFLOP/s
AMD Radeon 7970	32 x 64 Cores	per 64 Threads	925MHz	Yes	3.79 TFLOP/s
Intel Xeon E5-2690	2 x 8 Cores	256-bit SIMD	up to 3.8 GHz	No	486.4 GFLOP/s
Intel Xeon Phi 5110P	60 Cores	512-bit SIMD	1.053 GHz	Yes	2.02 TFLOP/s

Table 1: Hardware used for experiments

Device	Compiler	Achieved performance	Maximum (7 non-FMA + 6 FMA)
NVIDIA GTX Titan	CUDA 5.0	**2.63 TFLOP/s (80.1%)**	3.28 TFLOP/s
Intel Xeon E5-2690	Intel (13.1) - auto vect.	273 GFLOP/s (56.1%)	486.4 GFLOP/s
Intel Xeon Phi 5110P	Intel (13.1) - auto vect.	273.68 GFLOP/s (18.7%)	1.46 TFLOP/s
Intel Xeon E5-2690	Intel (13.1) - inner loop vect.	466.01 GFLOP/s (95.8%)	486.4 GFLOP/s
Intel Xeon Phi 5110P	Intel (13.1) - inner loop vect.	272.21 GFLOP/s (18.6%)	1.46 TFLOP/s
Intel Xeon E5-2690	Intel (13.1) - outer loop vect.	**476.1 GFLOP/s (97.8%)**	486.4 GFLOP/s
Intel Xeon Phi 5110P	Intel (13.1) - outer loop vect.	**1.26 TFLOP/s (86.3%)**	1.46 TFLOP/s

Table 2: Hardware specific implementations

Device	OpenCL	Achieved performance	Remarks
NVIDIA GTX Titan	1.1	2.19 TFLOP/s	Slight performance degradation
AMD Radeon 7970	1.2	1.98 TFLOP/s	Relatively good performance
Intel Xeon E5-2690	1.2	51.673 GFLOP/s	Auto-vectorization does not seem to work
Intel Xeon Phi 5110P	1.2	23.09 GFLOP/s	Auto-vectorization does not seem to work

Table 3: OpenCL port - GPU based (CUDA port)

Device	OpenCL	Achieved performance	Remarks
NVIDIA GTX Titan	1.1	1.96 TFLOP/s	The slowest version
AMD Radeon 7970	1.2	1.78 TFLOP/s	The slowest version
Intel Xeon E5-2690	1.2	292.04 GFLOP/s	Slower than auto vectorization
Intel Xeon Phi 5110P	1.2	203.6 GFLOP/s	Slower than auto vectorization

Table 4: OpenCL port - CPU based - inner loop manual vectorization

Device	OpenCL	Achieved performance	Compared to the best native version	Remarks
NVIDIA GTX Titan	1.1	2.45 TFLOP/s	**93.1% (CUDA)**	
AMD Radeon 7970	1.2	2.638 TFLOP/s	**133.2% (CUDA→OpenCL Port)**	
Intel Xeon E5-2690	1.2	391.87 GFLOP/s	**82.3% (manual vectorization)**	
Intel Xeon Phi 5110P	1.2	485.2 GFLOP/s	**37.2% (manual vectorization)**	w/ rsqrt
Intel Xeon Phi 5110P	1.2	1.17 TFLOP/s	**92.7% (manual vectorization)**	w/o rsqrt

Table 5: Final OpenCL version - CPU based - with outer loop manual vectorization

Technical Report No. UCB/EECS-2006-183, December 18, 2006

[8] Judit Planas, Rosa M. Badia, Eduard Ayguade and Jesus Labarta. Self-Adaptive OmpSs Tasks in Heterogeneous Environments. In proceedings of 27th IEEE International Parallel & Distributed Processing Symposium, IPDPS 2013

[9] `http://parlab.eecs.berkeley.edu/wiki/patterns`

[10] L. Chen, O. Villa, S. Krishnamoorthy, G.R. Gao, Dynamic load balancing on single- and multi-GPU systems. Parallel & Distributed Processing (IPDPS), April 2010

[11] C.S. de La Lama, P. Toharia, J.L. Bosque, O.D. Robles. Static Multi-device Load Balancing for OpenCL. Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on , vol., no., pp.675,682, 10-13 July 2012

[12] Aarseth, S. 2003. Gravitational N-Body Simulations. Cambridge University Press.

[13] Hubert Nguyen. 2007. Gpu Gems 3 (First ed.). Addison-Wesley Professional - Chapter 31. Fast N-Body Simulation with CUDA

[14] Verlet, Loup (1967). "Computer Experiments on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules" Physical Review 159: 98–103. doi:10.1103/PhysRev.159.98