MCMini: Monte Carlo on GPGPU

Ryan Marcus
`rmarcus@lanl.gov`

Summer 2012

Abstract

MCMini is a proof of concept that demonstrates the possibility for Monte Carlo neutron transport using OpenCL with a focus on performance. This implementation, written in `C`, shows that tracing particles and calculating reactions on a 3D mesh can be done in a highly scalable fashion. These results demonstrate a potential path forward for MCNP or other Monte Carlo codes.

1 Introduction

As a co-design application for exascale research and development[2], MCMini requires a sample implementation. Work from 2011[1] demonstrated a slower version of MCMini implemented in Python that could run using `CUDA`, `OpenCL`, `OpenMP`, `MPI`, and could also run in serial (without parallelism). This version of MCMini is focused on performance, and is thus written in `C`.

In order to create performance-capable code, MCMini only supports one set of parallel technologies: `OpenMP`, `MPI`, and `OpenCL`. Considerations for choosing these technologies are described in the next section.

2 Potential Exascale Technologies

2.1 A diverse playing field

Accelerator and co-processor technologies evaluated for MCMini include:

- GPGPUs utilizing NVIDIA's `CUDA`: NVIDIA's GPU language
- GPGPUs utilizing Khronos' `OpenCL`: An open standard for highly parallel computing
- The `Intel MIC` co-processor: One of Intel's HPC technologies

Each item was found to have various advantages and disadvantages which are documented in the following sections.

2.1.1 NVIDIA CUDA

CUDA is the original GPGPU language/API backed by NVIDIA. It features the ability to run a subset of C on the GPU. It is proprietary software which runs exclusively on NVIDIA devices, such as the Tesla.

2.1.2 Khronos OpenCL

OpenCL, like CUDA, features the ability to run a subset of C on the GPU. Unlike CUDA, OpenCL code can run on more than one platform, including NVIDIA GPUs as well as most AMD and Intel CPUs. It is an open standard backed by Apple and maintained by the Khronos Group.

It should be noted that Intel plans to add OpenCL support to their Intel MIC.

2.1.3 Intel MIC

Intel MIC is a new technology from Intel which is still under heavy development. The Intel MIC is a card containing a number of X86 processors that are similar to Intel's Xeon. The Intel MIC attempts to provide a highly-parallel environment that can execute Fortran and C code, and realistically, any code that can compile to its slightly modified X86/X86_64 instruction set.

While running current Fortran code seems like a very attractive option, the nature of the Intel MIC architecture may not allow it to attain the same level of performance as GPU devices. The X86 instruction set is very heavy, which increases power requirements. The memory architecture of the Intel MIC is identical to that of a standard CPU, which means developers will not be able to take advantage of faster, non-global memory. Running the same Fortran or C code may stifle transitions to more suitable parallel algorithms. OpenMP codes were designed to run using four to eight tasks, not fifty. MPI's memory overhead will probably decrease performance gains significantly [6].

2.2 Selection Criteria

2.2.1 Performance

All three technologies have shown very good performance under a large number of tests, and while different tests place different technologies ahead of each other, there does not seem to be a substantial difference between the optimal performance of each technology, especially between CUDA and OpenCL. The Intel MIC is still in very early development, but benchmark results are promising.

Some early (2010) benchmarks showed OpenCL code running 16% to 67% slower than CUDA code[4]. However, more recent benchmarks (2012) from Oakridge National Lab's SHOC (Scalable Heterogeneous Computing Benchmark Suite) have shown that OpenCL and CUDA performance have nearly equalized[5].

2.2.2 Hardware agnostic code

The “write-once, run-anywhere“ paradigm has been an important goal of some programming languages for a long time. Forming a language that is expressive enough (as well as a compiler that is smart enough) to map high-level concepts to a multitude of different devices is a difficult challenge.

Hardware-specific code, in the long term, is perhaps an even bigger challenge. If one is not lucky enough to have a single target device, maintaining code for many different platforms can be time consuming and costly.

The ever-evolving landscape of HPC and especially of exascale technologies amplifies this concern. Writing code for a platform represents a commitment to that platform, and since the platforms of exascale appear to be in constant flux, committing to a certain platform could be troublesome.

Therefore, writing code that can be used on a multitude of platforms provides a greater degree of security than platform-dependent code, even if there is a chance that the multi-platform code becomes obsolete. Because `OpenCL` code can presently run on CPUs, GPUs, and eventually the `Intel MIC`, it appears to be the best option in terms of hardware agnostic tools. `OpenCL` can even be extended to operate on entire clusters[3].

It should be noted that while the `Intel MIC` may appear to be hardware agnostic because code could be ran on CPUs as well as the `Intel MIC`, code compiled using the special instruction set of the `Intel MIC` could not be ran on any kind of GPU.

2.2.3 Open standard

Using an open standard is important to keep codes independent of vendors. If something were to happen to NVIDIA, `CUDA` code would likely become useless as the last generation of hardware produced by NVIDIA became deprecated.

Having an open standard also increases hardware agnosticism. When any vendor is allowed to produce a device conforming to an open standard, more than one vendor is likely to produce capable hardware.

2.2.4 Ease of use

Compared to the difficulties of creating parallel algorithms and writing efficient multi-threaded code, all platforms are fairly usable.

Toolsets for all three platforms are also well-developed. NVIDIA’s NSight, while running only on Windows, provides debugging and profiling support. AMD’s `OpenCL` implementation can be debugged using GDB and profiled using `OpenCL`’s standard profiling tools. The `Intel MIC` is compatible with the Intel and GNU toolchains.

Ease of use also depends on community support. Being able to search the Internet for an answer to a question can substantially increase developer productivity. While `CUDA` certainly has the largest user base at the moment, `OpenCL` is gaining traction. A community base for Intel’s `Intel MIC` has yet to appear.

2.3 Selection

Because of the importance of an open standard and the convenience that comes with code portability, OpenCL was selected for developing MCMini.

It should be noted that the selection of OpenCL as a platform says nothing about the optimal physical hardware. Selecting OpenCL as an HPC platform does not mean one is forced to purchase AMD GPUs. Because OpenCL is hardware agnostic, using it as a platform to write high-performance Monte-Carlo codes allows those codes to be ran on whatever hardware is optimal at the time. Another benefit of buying OpenCL compatible devices is that one gains access to two different platforms. For example, if one were to purchase Intel MICs, the Intel MICs could be used to run current codes until those codes have been ported to OpenCL, providing both an immediate benefit and a stepping stone to GPU based acceleration.

In order to support multiple devices, and to accelerate certain sections of code that are not computed on accelerator devices, OpenMP is used.

For clustering support (to reach multiple machines), MPI is used.

3 MCMini

3.1 Description

MCMini is a Monte Carlo mini-app designed as a performance-capable proof of concept for Monte Carlo neutron transport on GPGPU devices. The performance-sensitive nature of the code made the C language a natural choice.

MCMini traces particles through a mesh (described in the MCNP-compatible LNK3DNT format[7]) and calculates scatter, fission, and absorption reactions within cells containing mixed materials. MCMini traces any daughter particles down to a user-specified weight cutoff or predefined number of generations.

Compatibility between MCNP and MCMini is limited but functional. MCMini can read the LNK3DNT geometry description that MCNP utilizes. Many of the input decks for MCMini came from the DAWWG testing suite of MCNP.

MCMini is capable of scaling over multiple OpenCL devices, including CPUs and GPUs. MCMini can also scale over multiple nodes containing OpenCL devices through MPI.

3.2 Design

MCMini's algorithms and implementation were designed by Larry Cox and Ryan Marcus at the Los Alamos National Laboratory during the summer of 2011 and 2012. Previous work about vectorized Monte Carlo were influential[8].

A high-level description of the algorithms used can be found in the document "Developing a Monte Carlo mini-App for Exascale Co-Design," LA-XX-12-XXX. Implementation details (including many tricks and algorithm optimizations) can be found in the documentation accompanying the source code.

The physics involved in MCMini are modular, but the default implementation includes only simple Newtonian mechanics. No relativistic calculations or compensations are made, and the neutron cross-section data used was fabricated to create analytically-verifiable results.

The Newtonian physics used are cost-similar to the actual physics needed for accurate results, so performance benchmarks of MCMini are very relevant, and give a good preview at the potential for a full Monte Carlo neutron transport code running on GPUs.

3.3 Performance

MCMini's particle trace algorithm was designed to produce identical results to MCNP, and MCMini's attenuation algorithm, given a cross-section, is cost-similar to MCNP.

MCMini is capable of tracing a large number of particles through a very large mesh in a relatively small amount of time. The following table lists a few performance samples from MCMini, each one calculating fourth generation daughter particles.

Nodes	GPUs	Mesh cells	Particles	Time (m:s)
1	2	1000^3	10^5	0:50
2	4	2000^3	10^5	1:06
4	8	3000^3	10^5	1:51
8	16	10000^3	10^5	6:00

Table 1: MCMini times to trace particles through a large mesh

In addition to these high-level benchmarks, other considerations for HPC computation include CPU/GPU speed comparisons, multi-GPU scaling, multi-node scaling, and hardware comparisons. These are below.

3.3.1 CPU vs GPU Scaling

Because `OpenCL` code can run on both a GPU and a CPU, it makes sense to compare performance between the two.

The following benchmark was conducted on an `AMD Cypress` device and an `AMD Opteron(tm) Processor 6168` CPU. The problem ran was a `600 x 600 x 600` mesh containing a cone.

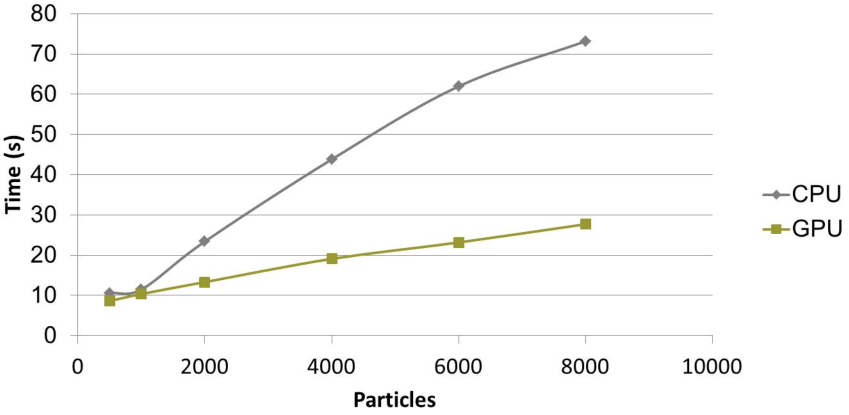
At a high number of particles, the GPU is over twice as fast as the CPU.

Both memory and speed affect this benchmark. Because the GPU has a much smaller amount of memory, a larger geometry may lead to substantial slowdowns because memory would have to be swapped between the host the GPU. None of the runs in this benchmark involved any swapping with host/system memory.

Particles	CPU Time (s)	GPU Time (s)
500	10.603	8.627
1000	11.466	10.355
2000	23.408	13.320
4000	43.818	19.091
6000	61.978	23.175
8000	73.161	27.706

Table 2: CPU v GPU times

Figure 1:
CPU vs GPU Times



3.3.2 Multi-GPU Scaling

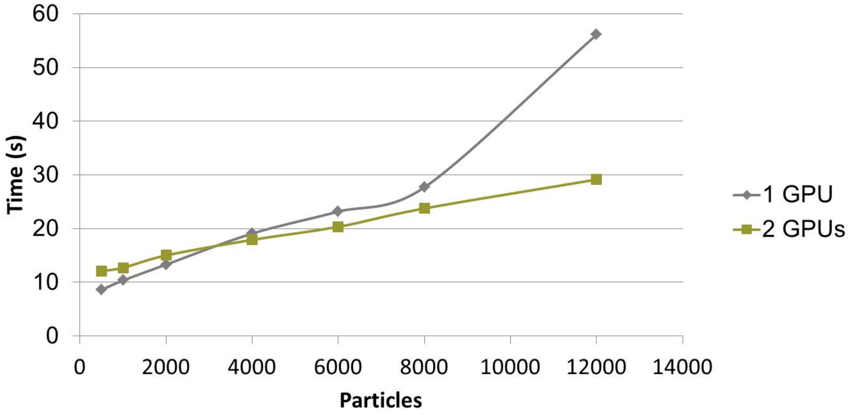
The `OpenCL` API allows one to run code on multiple `OpenCL` devices at the same time. MCMini can utilize multiple devices by dividing up work based on processor speed and memory capacity, doing all the partial calculations, and then reducing the results.

The following benchmark was conducted on two `AMD Cypress` devices. The problem ran was a `600 x 600 x 600` mesh containing a cone.

Particles	1 GPU Time (s)	2 GPU Time (s)
500	8.627	12.135
1000	10.355	12.72
2000	13.32	15.024
4000	19.091	17.921
6000	23.175	20.346
8000	27.706	23.763
12000	56.102	29.146

Table 3: 1 GPU vs 2 GPU

Figure 2:
Multi-GPU Times



As the number of particles grows, the two-GPU setup becomes nearly twice as fast as the single-GPU setup. This demonstrates very good linear scaling.

Since two devices are involved in the two-GPU setting, a much larger number of particles could potentially be traced, as much more memory is available. Every test in this benchmark, including the single GPU setup, did not require swapping with host/system memory.

3.3.3 Multi-node Scaling

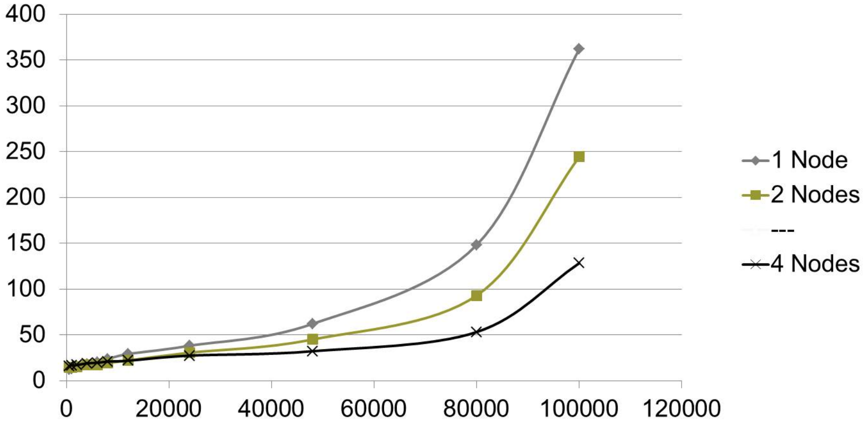
Using `MPI`, MCMini can run code over multiple nodes as well as multiple devices. MCMini divides up work to each node, then each node divides up work to each device. Each node reduces its data, and then the master node reduces all the other node's data.

The following benchmark was conducted on nodes with two `AMD Cypress` devices each. The problem ran was a `600 x 600 x 600` mesh containing a cone. `OpenMPI` was used for node-to-node communication on LANL's Darwin cluster.

Particles	1 Node	2 Nodes	4 Nodes
500	12.135	14.232	15.932
1000	12.72	14.455	16.851
2000	15.024	15.062	17.051
4000	17.921	16.982	18.998
6000	20.346	17.573	19.583
8000	23.763	19.149	20.899
12000	29.146	22.158	21.856
24000	38.184	30.553	27.366
48000	62.118	45.021	32.182
80000	148.235	93.054	53.157
100000	362.185	244.36	128.326

Table 4: 1 Node vs 2 Nodes vs 4 Nodes

Figure 3:
Multi-Node Times



MCMini appears to scale well, especially as the number of particles increases. Adding multiple nodes can clearly benefit large problems. There is no constraint that the number of nodes must be a power of two or an even number.

3.3.4 Hardware Scaling

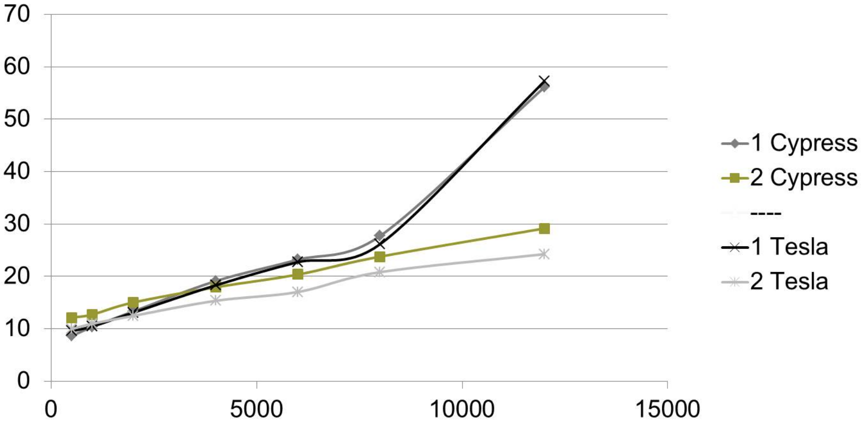
Because OpenCL code can run on many different types of devices, it makes sense to compare AMD hardware to NVIDIA hardware. An identical same codebase is used on both vendor's devices.

The following benchmark was conducted on nodes with two AMD Cypress devices each, and two NVIDIA Tesla C2070 devices each. The problem ran was a 600 x 600 x 600 mesh containing a cone.

Particles	1 Cypress	2 Cypress	1 Tesla	2 Tesla
500	8.627	12.135	9.576	9.955
1000	10.355	12.72	10.444	10.953
2000	13.32	15.024	13.086	12.468
4000	19.091	17.921	18.279	15.339
6000	23.175	20.346	22.718	17.019
8000	27.706	23.763	26.139	20.798
12000	56.102	29.146	57.22	24.24

Table 5: Cypress vs Tesla

Figure 4:
Cypress/Tesla Times



It should be noted that the AMD Cypress device was designed as a consumer-level video card, and the NVIDIA Tesla C2070 was designed as a compute device. It should also be noted that none of these tests exceeded the 1 GB memory capacity of the AMD Cypress device, so the full potential of the NVIDIA Tesla C2070 was not utilized.

The scaling and performance of both devices are similar, with the two-NVIDIA Tesla C2070 setup slightly out-performing the two-AMD Cypress setup.

3.4 MCMini / MCNP Comparison

In order to show that MCMini demonstrates the possibility of Monte-Carlo neutron transport on GPUs, results from MCMini need to, at a minimum, come close to those produced by MCNP.

3.4.1 Neutron flux calculations

At best, neutron flux calculations performed on identical geometries (mostly from the MCNP `DAWWG` testing suite) with MCNP can be called “slightly comparable“ to MCMini, but this was the goal.

MCMini’s results are similarly scaling to MCNP. As MCNP’s answer increases, MCMini’s answer increases. While MCMini’s results are rarely within the margin of error given by MCNP, the scaling factor between two problems ran in both MCMini and MCNP is approximately the same.

3.4.2 Time comparison

Any time comparison between MCMini and MCNP is inherently unfair because MCNP performs far more calculations than MCMini. However, geometries that take a hours to trace in MCNP, can be fully traced in MCMini in minutes or seconds.

4 Parallel Algorithms

The development of a performance-sensitive code like MCMini required several parallel algorithms, some of which are described here.

4.1 Random number generator

An important aspect of any Monte Carlo code is a good random number generator. In parallel setups, each thread needs to be able to access it’s own stream of random numbers at a very low cost.

4.1.1 Linear random number generators

Many GPU experts describe linear random number generators as insufficient for GPU-based tasks and prescribe several other, more complex methods[9]. The primary reason for the perceived inadequacy of linear random number generators is the limited period of 2^{31}. To get around this issue, MCNP uses a 64-bit linear random number generator that is very well suited to neutron transport[10]. Many other linear random number generators are also provided.

Double precision calculations on GPUs have been historically slow, normally cutting down performance by over 50%. Although both AMD[11] and NVIDIA[12] are releasing cards with over a teraflop of double-precision power, single precision still remains significantly faster.

Using MCMini with a double-precision random number generator does bring down performance substantially.

One trick used to substantially less the impact of the double-precision performance hit was to generate double-precision random numbers using an `int2`, an `OpenCL` vector type, and some bitwise tricks. The number is then mapped to an unsigned 32-bit integer.

A better solution was to intelligently utilize the limited period of a 32-bit generator. A Monte Carlo simulation only loses precision when the random number generator wraps around and a number is used twice for the same purpose. For example, consider a random sequence a_n. It is bad if a_2 gets used to determine two particle's initial X positions within a geometry because particles may not be sampled in an adequately random fashion. However, if a_2 gets used to determine the X position of a particle, and to calculate the energy deposited due to absorption by a different particle, there is not a problem.

One other solutions included in MCMini was to switch to a different linear random number generator after one had been exhausted.

4.1.2 Skip-ahead function

One necessity when using a linear random number generator on a GPU is a fast skip-ahead function. If each thread in a kernel needs to calculate all the random numbers leading up to that thread's starting point, performance will take a significant hit.

Because skipping ahead in a random number sequence is something that every thread is going to need to do at initialization whenever a thread requires a random number, it is important to have a very well optimized skip-ahead function.

An implementation of such a skip-ahead function relies heavily on bitwise operations, which are very efficient on GPUs[10]. Potential `OpenCL` code usable for a 32-bit generator is below. It was translated directly from the open-source MCNP random number generator. Replace `%MULT%`, `%ADD%`, and `%MASK%` with their appropriate values from the linear random number generator that is being used.

```
void skipAhead(int n, int* lastVal) {
        // assuming that n > 0
        int gen = 1, g = %MULT%, inc = 0, c = %ADD%;
        int nskip = n;
        int gp, rn;

        nskip &= %MASK%;
        while (nskip > 0) {
                if ((nskip >> 1) << 1 != nskip) {
                        gen = (gen*g) & %MASK%;
                        inc = (inc*g) & %MASK%;
                        inc = (inc + c) & %MASK%;
        }

        gp = (g+1) & %MASK%;
        g = (g*g) & %MASK%;
        c = (gp*c) & %MASK%;
        nskip >>= 1;
    }

    rn = (gen*(*lastVal)) & %MASK%;
    rn = (rn + inc) & %MASK%;
    *lastVal = rn;
}
```

4.2 Buffered Iteration Pattern

As described in previous works[8], it is often necessary to have a thread perform a task for a greater number of iterations than it's neighboring thread. For example, when tracing particles through a mesh, some particles will exit before others.

When doing something like tracing particles through a mesh, where each step results in some data (what cell the particles are in) and where some threads may finish before others, developers are left with two options:

- create a multi-dimensional array big enough to hold the maximum possible amount of data
- create a one-dimensional array big enough to hold a single iteration's worth of data, and copy that piece of data back to the host after each iteration

While option two may seem like it would use a lot of bandwidth, it can implemented using a double buffer, as described by the below pseudo-code. This algorithm reduces the space requirement of any such iterative algorithm from $O(n^2)$ to $O(n)$.

```
d = the data we are processing
b1 = an array big enough to hold the results of 1 step of d
b2 = another array big enough to hold the results of 1 step of d
*b = a pointer to an array, either b1 or b2

ll = a linked list to hold each step

allocate b1 on device
allocate b2 on device

b = &b1 // make b point to b1
```

```
while (iteration is required) {
        do_calculation_kernel(d, *b);
        t = copy *b from device to host
        ll.append(t)
        b = (b == &b1 ? b = &b2 : b = &b1);
        wait for any copy using *b
}
```

As long as the the `do_calculation_kernel` step requires less time than a copy, very little performance should be lost. If the `do_calculation_kernel` step does require more time than a copy, more buffers can be added, but this increases the space cost (although the space cost will still be far below $O(n^2)$).

If memory utilization is an issue, one could remove the double buffer and replace it with a single buffer, copy that buffer to the host each time, scan it for completed items, and then copy back a smaller array of items that have not been completed. This would greatly decrease performance.

Obviously, this pattern is only useful if one can read data from the device while also executing a kernel on the device.

4.3 Device Context Switching

When computing without an accelerator or co-processor, one either has enough memory, or one does not. When computing with an external device that has its own memory, memory management can become much more complicated.

For example, imagine a device with 8 units of memory, along with the following kernels and data. Assume that in order for the program to complete, all kernels must execute in order.

Kernel	Requires
1	a, b, c
2	b, c, d
3	a, b
4	c

Table 6: Example kernel requisites

Data	Size
a	4
b	2
c	2
d	1

Table 7: Example data sizes

When executing these kernels, one would have to switch between various contexts. When going from kernel 1 to kernel 2, one would need to free `a` from the device memory in order to fit `d`.

One might imagine a simple solution to this problem to be switching to the minimal context for each kernel. However, this might not be optimal, as it would require using additional bandwidth. Consider the following two runs of this program, one in which a minimal context is always preserved, and another where it is not.

Kernel	Minimum	Data	Bandwidth	Non-minimum	Data	Bandwidth
1	copy(a, b, c)	a, b, c	8	copy(a, b, c)	a, b, c	8
2	free(a) copy(d)	b, c, d	1	free(a) copy(d)	b, c, d	1
3	free(d, c) copy(a)	a, b	4	free(d) copy(a)	a, b, c	4
4	free(a) copy(c)	b, c	2	free(a)	b, c	0

Table 8: Minimal context and non-minimal context

Clearly, a smarter context manager is needed. The context manager would need to be able to look ahead at the memory requirements of future kernels, and determine the optimal route through the program to minimize bandwidth. Sometimes, this optimal route could be calculated before hand, but often the size of data is not constant, so a good context manager would need to do this at runtime.

The context manager used for MCMini is Oatmeal, which is described in a separate document included in the Oatmeal distribution.

5 Results

5.1 OpenCL

MCMini clearly shows that `OpenCL` is a viable HPC technology, and a potential route on the path to exascale. The massively parallel nature of GPUs and more modern CPUs is well-handled by the `OpenCL` standard. `OpenCL` continues to show signs of growth.

MCMini also demonstrates that `OpenCL` code can be effectively ran on both a CPU and a GPU, making it an ideal ”transition” technology. Pieces of code written in `OpenCL` will be able to run on both newer, GPU-equipped machines, as well as older, CPU-powered ones.

5.2 Monte Carlo

The speed and relative accuracy of MCMini shows that Monte Carlo methods can be easily adapted to GPUs, especially using `OpenCL`.

The mesh data structure proved to be a good data structure for the necessary 3D parallel operations.

Monte Carlo neutron transport methods generally have a very intuitive division of labor across multiple particles, which is naturally fitted to parallelism.

References

[1] Marcus, Ryan. ”Monte-Carlo Mini-App on Exa Framework” 2011. Cox, Larry. Marcus, Ryan. ”Python framework for co-design applications in exascale R&D” LA-UR-11-06086

[2] Cox, Lawerence. Marcus, Ryan. ”Co-design Applications for Exascale R&D.” 2011. LA-UR-11-06085

[3] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. 2012. SnuCL: an `OpenCL` framework for heterogeneous CPU/GPU clusters. In Proceedings of the 26th ACM international conference on Supercomputing (ICS ’12). ACM, New York, NY, USA, 341-352. DOI=10.1145/2304576.2304623 http://doi.acm.org/10.1145/2304576.2304623

[4] Kamran Karimi,Neil G. Dickson, Firas Hamze. 2012. A Performance Comparison of CUDA and OpenCL. `arXiv:1005.2581v3 [cs.PF]`

[5] Michael Feldman,February 28, 2012, ”OpenCL Gains Ground On CUDA” http://www.hpcwire.com/hpcwire/2012-02-28/opencl_gains_ground_on_cuda.html

[6] Steve Scott, ”No Free Lunch for Intel MIC (or GPUs)”. Apr 3 2012 http://blogs.NVIDIA.com/2012/04/no-free-lunch-for-intel-mic-or-gpus/

[7] Lawrence J. Cox, Ph.D. ”LNK3DNT Geometry Support: User Guidance for Creating and Embedding”, LA-UR-11-01654

[8] William R. Martin and Forrest B. Brown, ”Status of Vectorized Monte Carlo for Particle Transport Analysis.” The International Journal of Supercomputer Applications, Voulme 1, Number 2, pp. 11-32. 1987.

[9] Lee Howes, David Thomas. ”GPU Gems 3”, from the NVIDIA CUDA developer site, chapter 37. http://http.developer.NVIDIA.com/GPUGems3/gpugems3_ch37.html

[10] Forrest B. Brown, Yasunobu Nagaya. ”THE MCNP5 RANDOM NUMBER GENERATOR” American Nuclear Society, 2002. LA-UR-02-3782

[11] AMD 2008, ”Breaking the 1 Teraflop Barrier”. http://www.amd.com/cn/Documents/AMD_ds_isc_A4sm_061608.pdf

[12] Michael Feldman, ”NVIDIA Launches First Kepler GPUs at Gamers; HPC Version Waiting in the Wings” March 22, 2012 http://www.hpcwire.com/hpcwire/2012-03-22/NVIDIA_launches_first_kepler_gpus_at_gamers_hpc_version_waiting_in_the_wings.html?page=2