Considering GPGPU for HPC Centers: Is It Worth the Effort?

Hans Hacker[1], Carsten Trinitis[1], Josef Weidendorfer[1], and Matthias Brehm[2]

[1] Department of Informatics, Technische Universität München, Germany
{hacker,trinitic,weidendo}@cs.tum.edu
[2] Leibniz Rechenzentrum, Garching bei München, Germany
brehm@lrz.de

Abstract. In contrast to just a few years ago, the answer to the question "What system should we buy next to best assist our users" has become a lot more complicated for the operators of an HPC center today. In addition to multicore architectures, powerful accelerator systems have emerged, and the future looks heterogeneous. In this paper, we will concentrate on and apply the abovementioned question to a specific accelerator with its programming environment that has become increasingly popular: systems using graphics processors from NVidia, programmed with CUDA. Using three benchmarks encompassing main computational needs of scientific codes, we compare performance results with those obtained by systems with modern x86[1] multicore processors. Taking the experience from optimizing and running the codes into account, we discuss whether the presented performance numbers really apply to computing center users running codes in their everyday tasks.

1 Introduction

In recent years, the strategy for an HPC center to get best performance out of their users' codes has undergone significant changes. Ten to fifteen years ago, a great variety of processor types and systems existed, e.g. MIPS, SPARC, Alpha, or vector systems like Cray, NEC, Fujitsu, etc.. Today, the TOP500 list [1] (the list of the 500 fastest supercomputers in the world) is dominated by x86 Intel or AMD processor based systems with high speed interconnects like e.g. Infiniband [2], and only a small niche remains for other architectures, such as IBM's Power architecture. Every few years, the next generation of a parallel system or compute cluster was put into operation, consisting of faster general purpose processor architectures. For many users, recompilation was only needed if the new system brought a change in the processors' Instruction Set Architecture (ISA).

Today, the number of options to choose from for a future system has increased again. On one hand, this is driven by technical boundaries in processor design, leading to multicore architectures with a wide variety of design options even inside a chip. Currently available multicore processors are often not best suited

[1] By x86 we also refere to the 64-bit extension of the x86 ISA.

to characteristic requirements of scientific code, e.g. only raising computational power but neglecting data transfer between components often results in reduced scalability for parallel code. On the other hand, the computer gaming industry has put a lot of effort into developing specially tuned graphics hardware devices, which get more similar to standard processors with every new generation [3]. The capability of graphics processing units (GPU) has reached a stage where they seem general enough to be useful to areas other than graphics, establishing the term GPGPU (General Purpose Graphics Processing Units) [4,5,6]. Companies such as NVidia or ATI/AMD see the HPC community as potential customers. Actually, the pure computational power achievable with tuned codes fitting a GPU's hardware capabilities is astonishing, even more so as this hardware is quite cheap since it is build for the mass market. However, graphics hardware traditionally is programmed in a way significantly different from regular software programming. It uses interfaces tuned for graphics requirements such as OpenGL and MS Direct3D. Due to this fact, GPU manufacturers introduced new proprietary programming models and according SDKs (e.g. NVidia CUDA [7], ATI Stream [8]).

While the available accelerator systems are attractive, a closer look should be taken at the requirements of scientific/HPC codes to judge usefulness. Ranging from requirements for IEEE floating point compliance and high bandwidth for data transfer, to stability and reliability, there is still room for improvement. Hence, adaption of accelerator hardware as part of the computer equipment of an HPC computing center has multiple aspects. From a purely operational point of view, these are:

– Improvement regarding *performance/cost* ratio in contrast to standard hardware: The cost that has to be taken into account must include power consumption and cooling needs both for GPU accelerators and standard systems, respectively.
– Expected *exploitation* of accelerator systems: What is the best way to fully utilize resulting heterogeneous systems? How many users are expected to actually run code which can utilize the accelerator parts?
– *Stability/reliability* of hardware and software, e.g. device driver, for use in multiuser environments.

To be able to answer the second aspect above regarding user acceptance of accelerator systems, it should be kept in mind that, besides performance, their main concerns are:

– *Ease of use*: What amount of studying effort is required to make optimal use of hardware? Is there a need for rewriting parts of the code from scratch, or can existing C/Fortran code be incrementally tuned to use additional features? Problems for adaption for the user could comprise a new programming model, new languages and new tools to be used. Similar, if the hardware does not completely conform to IEEE floating point standards (exactly same result, exception support), checking the correctness of a port can require a significant amount of time.

– *Persistency* and long-term support: Is it worth for users to tune their code for a given accelerator system? If the system involves a new programming model/-language, this should be part of a standardization process.

In this paper, we concentrate on the reachable performance of graphics processors from NVidia using CUDA. In order to achieve this, we have tuned a set of codes defined within work package (WP) 8 in the PRACE project. The work presented here was carried out in the context of the EU PRACE project,which prepares the creation of a persistent pan-European HPC service, consisting of several tier-0 centers providing European researchers with access to capability computers and forming the top level of the European HPC ecosystem. PRACE is a project funded in part by the EU 7th Framework Programme [9]. If available, we also measured optimized library codes with similar functionality. The results were compared with numbers obtained from systems consisting of recent x86 processors. In addition to performance results, we take a look at the usefulness of employing NVidia hardware as part of the computer equipment in an HPC center, taking the aspects presented above into account. In the process of optimizing and running the code, we obtained sufficient experience to be able to give profound arguments for discussion.

The paper is structured as follows: In the next section, we give an overview of the broader scope of PRACE where this work was done, and present some related work. Then, the programming model of CUDA is shortly presented. Afterwards, the different codes from the benchmark are shown, followed by optimization and tuning tips relevant for the codes. The next section presents performance numbers in detail. Finally, we discuss the advantages and disadvantages of using such accelerator systems at all, before we give a conclusion and an outlook on future work.

2 Related Work

In this paper, we concentrate on CUDA. However, the porting of the benchmarks was carried out as part of the "technology watch" subproject of PRACE with a broader focus. There, all kind of different architectures relevant for computing centers are taken into account, as can be read in the respective project deliverables [10]. The PRACE benchmarks represent simple kernels quite easily to port to various architectures, allowing for evaluation in the tight time frame of the project, i.e. striving for medium term statements. The Rodinia benchmark suite [11] takes a different approach. It specifically emphasizes on heterogeneous computing, fusing multicore architectures with GPU components. Further, it tries to cover the "dwarfs" application categorization of Berkeley [12]. This approach could become relevant for follow-up subprojects of PRACE.

There are quite a few projects world-wide that work towards the next-generation supercomputer, and collaborate in the International Exascale Software Project (IESP) [13]. The web page provides presentations and reports about the current status of these projects. While we expect that accelerators such as

GPUs to play an important role in this context, concrete statements are still pending.

There are a lot of papers presenting ports of all kind of applications to GPUs (see gpgpu.org [4]). However, it is quite difficult to estimate how much of these research works will result in code to be used by users of an HPC center.

3 CUDA as an Example for GPGPU Programming

In recent years, outsourcing compute intensive vector operations to graphics cards has become a popular pastime activity, especially in numerical simulation. Many people achieved quite good performance and scalability on non trivial problems by the so called Compute Unified Device Architecture (CUDA), a parallel programming model developed by NVidia. CUDA is an extension to the C programming language and allows programmers to utilize NVidia graphics cards for scientific computations. Each CUDA program consists of a so called *host* section and the *device* sections. The *host* section is executed sequentially and calls one or more so called *kernels*. These kernels are executed as threads in parallel on a *device*. Threads are grouped into blocks with up to three dimensions, where threads within the same block can be synchronized. The maximum number of threads per block is 512.

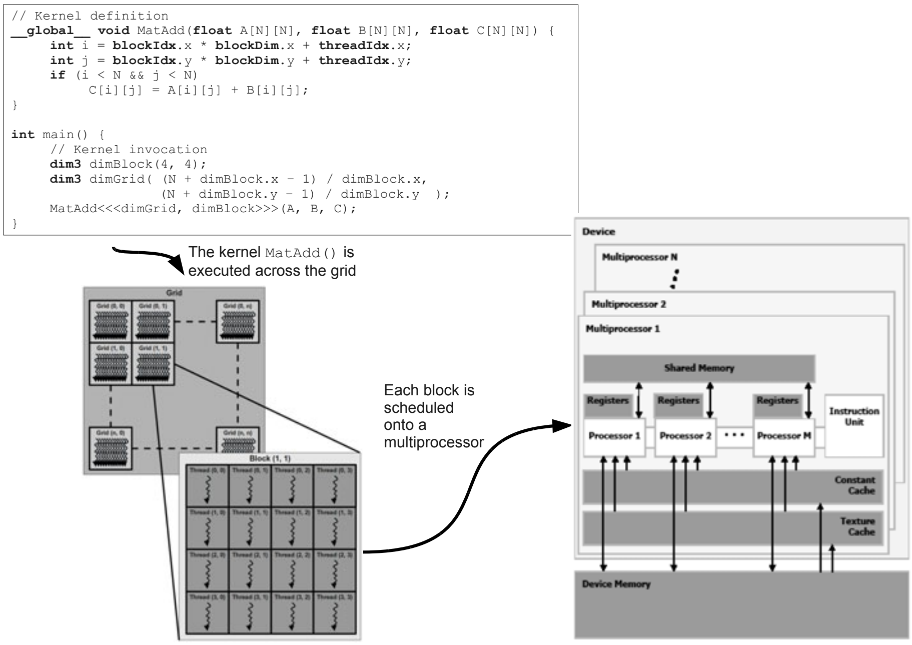
Thread blocks are grouped in one- or two dimensional *grids*. A grid may contain up to 65535 blocks per dimension. Thread blocks are distributed over the available SMs (Streaming Multiprocessors) upon execution. Each SM can serve up to 1024 coexisting threads. Current NVidia hardware contains up to 30 SMs; with each block being executed on exactly one SM. An SM consists of eight scalar cores with 2048 32-bit registers and 16 KB shared memory. Each of these cores is capable of performing integer and single precision floating point operations. Upon execution, blocks are divided into so called *warps*. CUDA currently groups 32 threads into *warps*. Within one *warp*, threads are always synchronous. Thread creation and scheduling are completely done in hardware.

Figure 1 shows a CUDA program for matrix addition ($C = A + B$). Each thread computes exactly one value of the result matrix. Blocks of 4x4 threads each are created and grouped in a two dimensional grid. Blocks are then executed by multiprocessors.

4 Methodology

4.1 The PRACE WP8 Benchmarks

In order to be able to assess and compare the performance among the vast amount of mainly heterogeneous PRACE prototypes, it was decided to select a set of small, easy to port, yet meaningful computational kernels. As a guideline for the selection, the principle of 'dwarfs' [14][12] was consulted. A dwarf is an



```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
     int i = blockIdx.x * blockDim.x + threadIdx.x;
     int j = blockIdx.y * blockDim.y + threadIdx.y;
     if (i < N && j < N)
          C[i][j] = A[i][j] + B[i][j];
}

int main() {
     // Kernel invocation
     dim3 dimBlock(4, 4);
     dim3 dimGrid( (N + dimBlock.x - 1) / dimBlock.x,
                   (N + dimBlock.y - 1) / dimBlock.y  );
     MatAdd<<<dimGrid, dimBlock>>>(A, B, C);
}
```

Fig. 1. CUDA example with mapping to hardware

algorithmic method that captures a pattern of computation and communication. The underlying numerical methods may change over time, but the claim is that the underlying patterns persist through generations. Three important dwarfs are namely dense linear algebra, sparse linear algebra and spectral methods.

Due to its modularity, the codes from the EuroBen benchmark suite[2] were chosen. This benchmark suite provides synthetic programs for scientific and technical computing. It is organized in modules of increasing complexity. Module 1 is concerned with the performance of important basic computational kernels like the dot-product or the `axpy` operation. The second module considers basic numerical algorithms that use the operators from module 1. These include matrix-vector multiply, solution of full and sparse linear systems or FFTs, etc. In the 3rd module skeleton applications like ODE and PDE solvers are tested that in turn use the algorithms from module 2.

As a representation for dense linear algebra, mod2am was selected. This is a dense matrix-matrix multiplication in the form of $C = AB$. In the area of sparse linear algebra, the choice fell on mod2as, a sparse matrix-vector product in the form of $c = Ab$. The matrix is stored in the compressed row storage (CRS) format. Finally mod2f, a 1-D complex FFT represents the dwarf spectral methods.

[2] http://www.euroben.nl

4.2 Porting and Optimization Strategies

mod2am. The obvious way to implement this benchmark is to apply the library calls `cublasDgemm` / `cublasSgemm`. As with every accelerator-card, one needs to transfer the data to the cards' main-memory. CUDA offers various ways to copy data. For this application, two methods were chosen. The first method is to allocate the host data-structures with malloc and the same data structures on the GPU and then copy the data. However, the MMU is involved in every page-request. The second method allocates the host data-structures via a special CUDA call (`cudaMallocHost`) which marks the allocated pages as non-swappable and allows data transfer to take place without the interference of the MMU. Since the mod2am-benchmark uses a typical C-'row-major' data arrangement, the transpose option for the input matrices was used. Unfortunately, there is no transpose option for the output so one has to write a kernel that transposes the matrix. There is an example code for a matrix transposition in the SDK that was modified accordingly. The transposition is done out-of-place by blocking the matrix, transposing the block and storing it back to the main memory of the card. NVidia gives a small example of a blocked dense matrix-matrix multiply in the programming guide. For comparing to CUBLAS, this kernel was also implemented. As the original NVidia example kernel only allowed matrices with the multiple size of the blocks, a few changes (mainly if clauses) had to be applied to deal with matrices of arbitrary size. Due to the existence of CUBLAS, the porting of this benchmark was straightforward. Only the different data arrangement (column-major vs. row-major) required slightly more effort.

mod2as. NVidia offers no library for sparse matrix operations. However, on CUDA Zone [7] the publication 'Efficient Sparse Matrix-Vector Multiplication on CUDA' by Nathan Bell and Michael Garland [15] can be found. The proposed CSR-kernel was modified to fit the needs of the C reference implementation of mod2as. It was not necessary to change the data structures. The CSR-kernel exploits the use of the shared memory within the Streaming Multiprocessor. Because of the warp concept (a warp consists of 32 threads that are all synchronous) a very efficient reduction can be performed. Furthermore, this kernel puts the x-vector into the texture-cache of the GPU. Since the x-data is reused multiple times this improves efficiency.

mod2f. By the time when the porting of mod2f has been started, cuFFT (the NVidia FFT implementation) only supported single precision. The single precision port therefore was straightforward. For double precision, a port of the given Radix-4 FFT C-code was carried out.

The given code works out-of-place, which means that additionally to the input vectors (real and imaginary), two other vectors are used for intermediate values. It also uses a precalculated vector of sines and cosines. The sine/cosine vector is being calculated by two nested loops that leads potentially to a major load-imbalance. However, as the loops form a geometric sequence, they can be unrolled. Thus, each thread calculates exactly one element of the result.

The calculation consists of multiple Radix-4 rounds plus additional Radix-2 rounds if the input data is not a power of 4. The Radix-4 rounds always use four real and four imaginary values, resulting into eight values to be calculated. Since a SM has eight scalar units and a warp consists of 32 threads, this fits perfectly (4x 8 threads).

As a GPU does not have caches like a CPU, access to the global memory is very slow (400-600 cycles). To hide this latency, NVidia recommends starting between 64-192 threads (oversubscribing). Additionally the memory accesses of a half-warp (16 threads) are coalesced in bundles of 128/64 or 32 bytes. In order to minimize memory access, the data types double2/float2 (struct of two doubles/floats) should be used. Each round always requires both the real- and the imaginary value. The .x value is the real- and the .y value is the imaginary-part of the vector/number. To fully exploit the usage of the double2/float2 type one can use the texture cache. This loads both the .x/.y-values to the cache. Hence, the .y-value read is almost for free. The readability of the code suffers considerably by using the cache. However, by using float2/double2 and texture-cache, the accesses to global-memory could be halved (9-10% speedup).

There are also many (integer) intermediate values that are calculated at the beginning of each round. Each SM is able to perform eight (integer) additions or bit-operations each clock-cycle. However, an SM can only perform two 32-bit multiplications per cycle. Therefore, one can use the `__mul24/__umul24` intrinsic (8 instructions / cycle). Both inputs require a value less or equal to 24 bit. The result is 32 bit. The first implementation also calculated those values ahead and stored the results in the shared memory of the SM. The use of the CUDA profiler indicated that it is faster (5-6%) to have each thread calculate the value itself and store the results in registers.

The double precision port was written generically and can calculate single precision as well. However, it turned out that a few kernels had to be specialized. The first implementation used two intermediate values, which were stored in the shared memory of the SM. This led to bank-conflicts with double precision. For single precision, because of being only 32-bit wide, this works perfectly. Furthermore, for double precision, the intermediate values are substituted twice in order to have only one calculation (the intermediates are stored in registers). If using single-precision the usage of 'fused multiply-add' (just cuts the result of the multiply instead of rounding) results in slightly different values than without. One can use an intrinsic to avoid 'fused multiply-add' (compiler-default). In order to avoid the rounding errors a considerable coding overhead is needed. Two FFT-rounds require two distinct loop variables. Here the grid-feature of CUDA can be used. One can start the blocks of calculation in a grid of up to two-dimensions. Each dimension is used as a loop variable.

The optimal number of threads per block is 64. 32 threads are not sufficient to hide the latency of the global memory (400-600 cycles). With more than 64 threads, the overhead of the calculations outweigths the latency hiding.

5 Results

The CUDA results were obtained on the PRACE prototype 'uchu'. A node consists of a Intel Harpertown (Xeon E5462 at 2.8 GHz) with 16 GB of system memory and two NVidia Tesla C1060 cards each with 4 GB of GDDR3 memory. The cards are attached via PCIe x16 (gen2) with a theoretical maximum throughput of 8 GB/s. However the measured maximum is only 5.7 GB/s. The installed NVidia/CUDA driver version was 190.18 and the toolkit version 2.3.

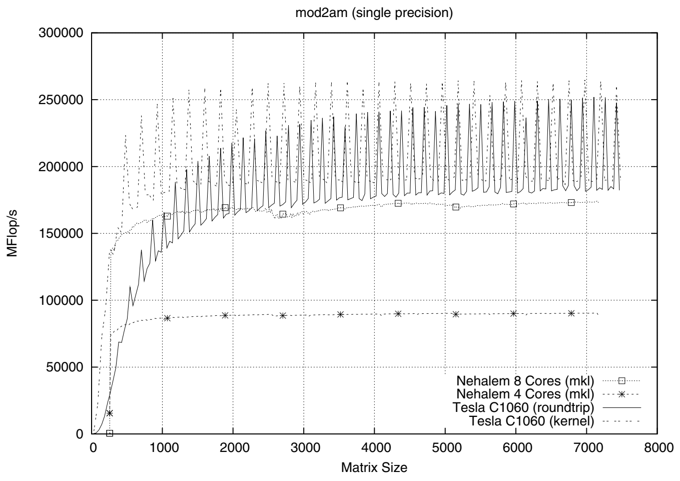


Fig. 2. mod2am in single precision

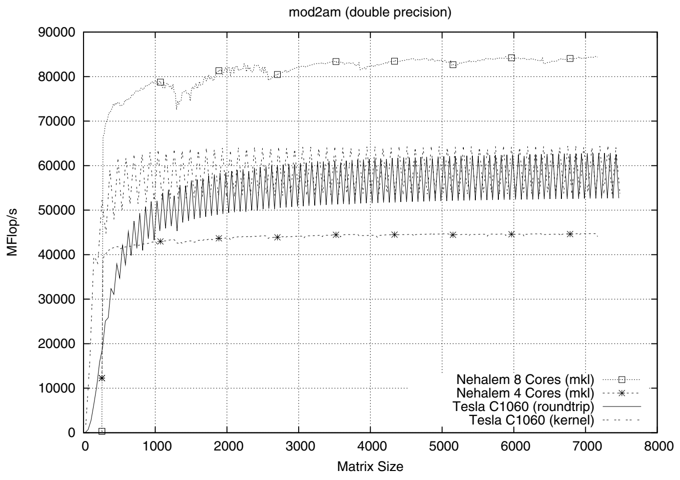


Fig. 3. mod2am in double precision

The comparison results were obtained on the SGI Altix ICE System (another PRACE prototype). Each node consists of two sockets, each with a Intel Nehalem EP (at 2.53 GHz). The node has 16 GB of system memory. The benchmarks were compiled using the Intel `icc` in version 11.1.064 and the Math Kernel Library (MKL) in version 10.1.

Figure 2 and Fig. 3 show the results of `mod2am`. Due to its highly parallel nature, this kernel is the sweet-spot for GPGPUs. The observed jitter in the CUDA results is due to underutilization of the hardware because of the input

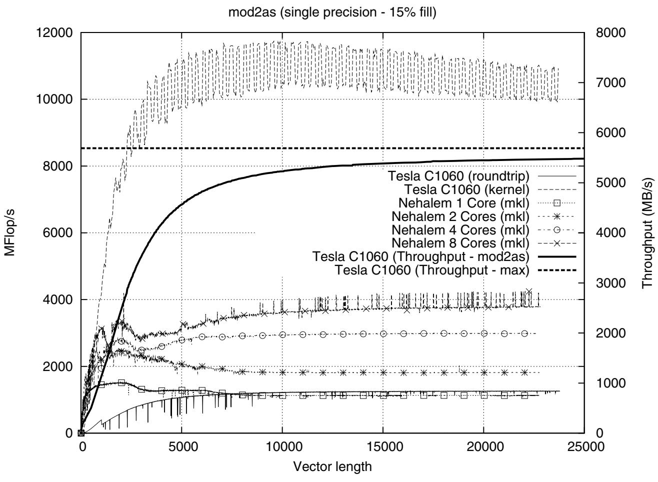


Fig. 4. mod2as in single precision

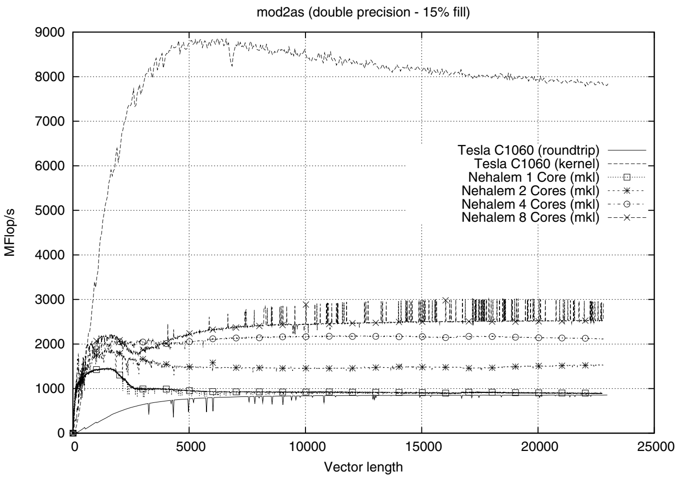


Fig. 5. mod2as in double precision

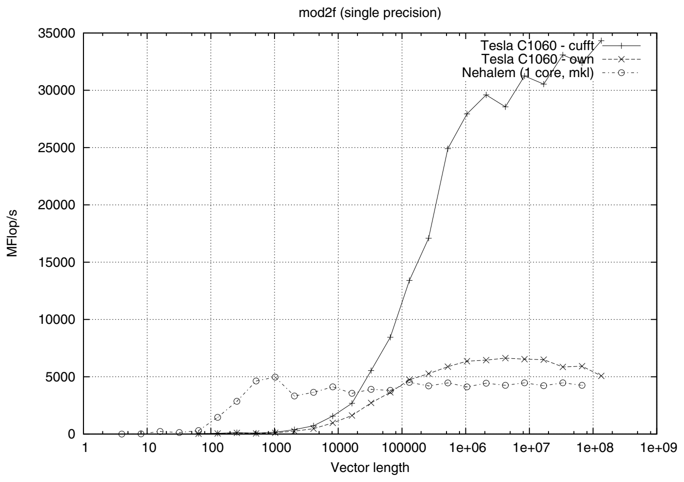


Fig. 6. mod2f in single precision

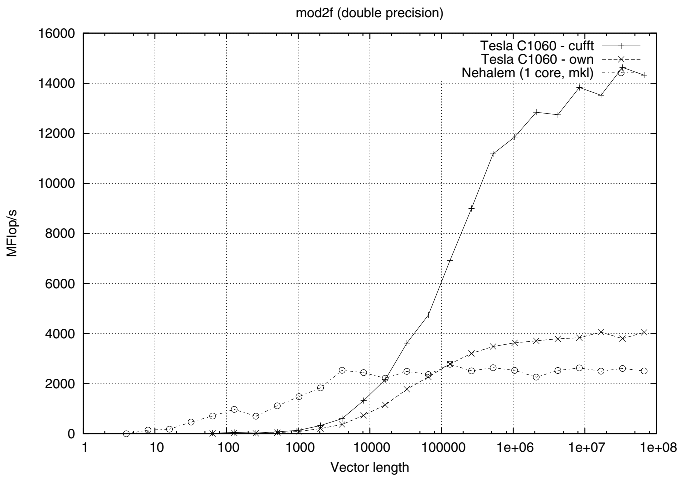


Fig. 7. mod2f in double precision

data size. The two different CUDA curves show achieved performance of the pure kernel and the performance with the inclusion of the data transfer (roundtrip).

Figure 4 and Fig. 5 show the results of `mod2as`. The main issue with this benchmark is that the data-transfer as well as the calculation are both of complexity O(n). The overall performance is limited by the throughput of the PCIe bus as shown in Fig. 4. The horizontal line (”throughput max”) indicates the maximum PCIe throughput on the ’uchu’ system. The tangential line shows the measured throughput of `mod2as`. The performance numbers for the pure kernel, with the data already on the card, indicate the potential of the Tesla card.

Therefore the suitability here clearly depends on whether the data can stay on the accelerator and be reused for multiple iterations or if the results have to be processed outside the accelerator after each calculation.

Figure 6 and Fig. 7 compare the results of `mod2f` with cuFFT, a hand coded version of a radix-4 FFT and MKL on Nehalem. The MKL provides only a serial version for the 1D-FFT. Recently NVidia released a new version of cuFFT (v.2.3) which supports double precision. The single precision performance was enhanced substantially. The cuFFT versions are an excellent choice for big inputs.

6 Discussion

Taking into account the results obtained by our benchmarks, this section will discuss the issues pointed out in the introduction.

From the HPC center's point of view, the following conditions apply:

– *Performance/Cost*: In terms of acquisition costs, graphics cards turned out to be an extremely cheap option for HPC, but in terms of operating costs, it must be taken into account that these nodes might be idle for a non-negligible part of their lifetime, still consuming additional power for cooling etc. . Therefore, future GPGPU accelerators should allow turning off idle parts of the system in order to guarantee efficient use.
– *Exploitation*: Despite the fact that only few users are willing and able to optimize their code the way described in this paper, there has been a strong demand for computing centers to offer compute nodes equipped with the respective hardware. However, as this applies only to a fraction of the users in a computing center, only a fraction of cluster nodes should be equipped with high end graphics cards for HPC usage.
– *Stability/reliability* of hardware: At the time of writing, existing graphics cards do not support error correcting codes (ECC), as this is not an issue in graphics programming - if one of several million pixels in an image has the wrong color, the user will not notice. DRAM failures are discussed in detail in [16]. Even if not all memory failures can be corrected, it is of major importance in an HPC Center to at least detect failing DRAMs. However, NVidia has announced that its new Fermi card will support ECC.

From the users' point of view, the situation looks as follows:

– *Ease of use*: Comprehensive study is certainly involved for users and/or programmers who intend to port their applications to CUDA. However, CUDA is relatively easy to learn for experienced programmers, as web pages provide plenty of examples. From a computing center's point of view, it would be a lot easier for users to upgrade to hardware that supports existing standards like e.g. OpenMP, PThreads, etc. or libraries like e.g. MPI. Programs should still run in standard C, C++, or FORTRAN environments without major modification, i.e. downward compatibility should be maintained to avoid unnecessary porting overhead. Regarding libraries, CUBLAS provides almost the complete set of BLAS routines for numerical operations. However, when it comes to more complex operations like e.g. sparse matrix computations, the user has to implement the underlying numerical algorithms. Also, current GPGPUs are not yet fully IEEE compliant, which can lead to errors when carrying out floating point operations. However, this might change with future GPGPU generations.

– *Persistency*: From our point of view, it is hard to foresee if a programming environment like CUDA, which is heavily derived from proprietary NVidia hardware (see section 3), will still prevail in five years. The upcoming standard for GPGPU programming, OpenCL [17], at least ensures that code is not bound to the success of only one vendor.

7 Conclusions and Future Work

In this paper, the practicability of using GPGPUs as accelerator cards in HPC centers has been investigated. We used several benchmarks from the PRACE WP8 benchmark suite and implemented these under CUDA on contemporary NVidia hardware. The benchmark results were compared with contemporary standard x86 based systems. In the discussion section, the issues raised in the introduction were answered according to the obtained results. We came to the conclusion that, at the time of writing, graphics cards are not yet a suitable alternative to "standard" HPC architectures, as several issues from both the computing center's and the users' point of view indicate that the effort is still too much. However, the picture might change with future GPGPU architectures like e.g. NVidia Fermi, which will be subject to future investigations.

While one can argue whether results from three numerical kernels are sufficient to draw any conclusions regarding the benefit of GPU usage in computer centers, the same procedure was chosen within PRACE for comparing different architecture types. In future investigations additional benchmark kernels should be examined to cover different types of applications, e.g. based on the dwarfs classification [12].

Acknowledgments

This work was financially supported by the PRACE project funded in part by the EU's 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-211528. We specially thank our colleague Iris Christadler for valuable input from the HPC center's view.

References

1. Top500 Consortium: The Top 500 supercomputing sites, `http://www.top500.org/`
2. Infiniband Trade Association: Infiniband Interconnect Homepage, `http://www.infinibandta.org/`

3. Novakovic, N.: CPU and GPU now, the convergence goes on. The Inquirer (October 2009), `http://www.theinquirer.net/inquirer/opinion/1560330/cpugpu-convergence-goes`
4. GPGPU.org: A central resource for GPGPU news and information, `http://gpgpu.org`
5. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum 26(1), 80–113 (2007)
6. Harris, M.: Mapping computational concepts to GPUs. In: ACM SIGGRAPH 2005 Courses. ACM Press, New York (2005)
7. CUDA Zone: The resource for CUDA developers, `http://www.nvidia.com/object/cuda_home.html`
8. Advanced Micro Devices, Inc.: ATI Stream Software Development Kit (SDK), `http://developer.amd.com/gpu/ATIStreamSDK`
9. PRACE: Partnership for Advanced Computing in Europe, `http://www.prace-project.eu`
10. PRACE: Public deliverables, `http://www.prace-project.eu/documents/public-deliverables-1`
11. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A Benchmark Suite for Heterogeneous Computing. In: Proceedings of the IEEE International Symposium on Workload Characterization (IISW). IEEE, Los Alamitos (October 2009)
12. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: a view from berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley (December 2006)
13. IESP: International exascale software project homepage, `http://www.exascale.org/`
14. Colella, P.: Defining software requirements for scientific computing (2004)
15. Bell, N., Garland, M.: Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation (December 2008)
16. Schroeder, B., Pinheiro, E., Weber, W.D.: DRAM errors in the wild: a large-scale field study. In: SIGMETRICS 2009: Proceedings of The Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, pp. 193–204. ACM, New York (2009)
17. Khronos Group: OpenCL - The open standard for parallel programming of heterogeneous systems, `http://www.khronos.org/opencl/`