A comparison of CUDA and OpenACC: Accelerating the Tsunami Simulation EasyWave

Steffen Christgau, Johannes Spazier, Bettina Schnor
Institute of Computer Science
University of Potsdam
August-Bebel-Straße 89
14482 Potsdam
{*christgau, spazier, schnor*}@*cs.uni-potsdam.de*

Martin Hammitzsch, Andrey Babeyko, Joachim Wächter
GFZ German Research Centre for Geosciences
Telegrafenberg
14473 Potsdam
{*martin.hammitzsch, babeyko, wae*}@*gfz-potsdam.de*

***Abstract*—This paper presents an GPU accelerated version of the tsunami simulation EasyWave. Using two different GPU generations (Nvidia Tesla and Fermi) different optimization techniques were applied to the application following the principle of locality. Their performance impact was analyzed for both hardware generations. The Fermi GPU not only has more cores, but also possesses a L2 cache shared by all streaming multiprocessors. It is revealed that even the most tuned code on the Tesla does not reach the performance of the unoptimized code on the Fermi GPU. Further, a comparison between CUDA and OpenACC shows that the platform independent approach does not reach the speed of the native CUDA code. A deeper analysis shows that memory access patterns have a critical impact on the compute kernels' performance, although this seems to be caused by the compiler in use.**

I. Introduction and Motivation

Within the EU-co-funded project *Collaborative, Complex and Critical Decision-Support in Evolving Crises Project (TRIDEC)* an early warning system for tsunamis is developed [1]. In case of a seismic event, the warning center has to evaluate the probability, the dimension and the locality of a potential tsunami based on the gathered sensor data. One of the core components of TRIDEC is the simulation *EasyWave*. It uses the sensor data as well as topology and bathymetric information and computes characteristics of the tsunami, e.g. the wave heights and coastal impact times in the affected regions.[2] A visualization of EasyWave's output is shown in Figure 1.

As computational time is a critical aspect in its use-case, EasyWave should execute as fast as possible. Modern GPUs offer capabilities to solve massively parallel problems in short times. Further, they are achievable and can be easily integrated within the compute server of the Early Warning System. Therefore, GPUs have been chosen as target platform for the parallel version.[1]

For the parallelization of EasyWave, there were different programming models available. Since our test systems were equipped with NVIDIA cards, a native CUDA [4] implementation was one option. Another choice was the use of OpenCL [5] or OpenACC [6]. While CUDA is tied to NVIDIA's GPUs, the OpenCL standard addresses different vendors and hardware platforms (GPUs, CPUs, Clusters etc.). Further the OpenACC standard resembles the OpenMP approach for shared memory programming: With few to no changes in the original source, compiler directives gives hints on the automatic translation of the sequential source code in a parallel version for the according hardware architecture. In case of OpenACC, the target platform are hardware accelerators like GPUs.

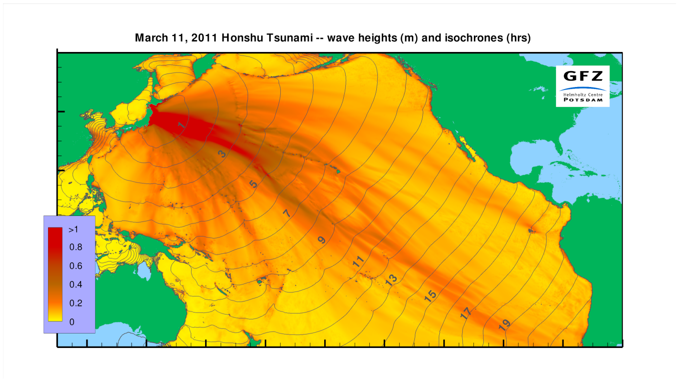

March 11, 2011 Honshu Tsunami -- wave heights (m) and isochrones (hrs)
GFZ
>1
0.8
0.6
0.4
0.2
0
1
3
5
7
9
11
13
15
17
19

Figure 1. Visualization of the Honshu Tsunami in March 2011 showing the heights and propagation of the tsunami wave. [3]

A comparison of the effort-performance ratio of OpenCL and OpenACC for two real-world applications was given in [7]. The authors conclude from their experiments that OpenACC offers a promising ratio of development effort to performance. The experiments were done on a NVIDIA Tesla C2015 with an early OpenACC implementation by Cray. The OpenACC implementation shows 80 % resp. 40 % of the OpenCL performance, depending on the application.

Since a native CUDA implementation should result in the best performance, we decided to use CUDA in the first step. In the following, the CUDA version of EasyWave was tuned using different techniques to exploit the capabilities of the GPU hardware. Since those optimizations tend to be hardware specific, the expected improvements on the performance was verified on different GPU hardware available in the two participating research institutes.

On one hand, these optimizations have improved the speedup of the application, but on the other hand they were time-consuming and needed knowledge about the underlying hardware. Additionally, they are likely to be specific to a certain generation of the compute hardware. This requirement makes it difficult for a non-expert programmer to easily write efficient code. This gave us motivation to implement also a parallel GPU version based on OpenACC which allows to mark code to be offloaded to accelerator hardware, e.g. GPUs.

The remainder of the paper is organized as follows: Section II and III give an overview over the sequential version of EasyWave and the experimental environment. In the following section, the design of the CUDA implementation is presented and different performance optimizations are discussed. Section V presents the OpenACC variant of the accelerated Tsunami simulation, followed by the conclusion.

II. Tsunami Simulation EasyWave

A. Sequential Version

The sequential version EasyWave [8] is written in C++ and uses a grid that represents the geographical area potentially affected by a tsunami. In most cases, the grid uses two dimensions with a granularity of two arc minutes. Usual scenarios of real-world incidents simulated with EasyWave have dimensions of about 2800×1800 grid points. For each grid point, the current and maximum wave height as well as physical water fluxes are stored in separated single precision floating point arrays. The pointers to these arrays are organized in another array leading to an array of pointers.

Due to the time-critical aspects of the simulation, simplified computational means like linear approximations are used [9]. Further, the physical aspects of wave propagation allow the usage of a window that restricts the computation: Only grid points within the window have to be computed. At the end of a single time step, i.e. after computing wave heights and fluxes for all points within the window, the borders of the window are analyzed to determine the need for an expansion of the window, which would be extended dynamically in that case. It has to be noted that data dependencies exist between the computation of fluxes and wave heights.

Within the window, the update of the wave height is done with a three-point stencil computation. It uses the upper and the left neighbor of the current cell as well as the current cell itself (TLC-stencil). Similar, the update of the fluxes in each grid point is done with a three-point stencil as well, but uses the right and the lower neighbor (BRC-stencil). In any case the update is done by iterating over the horizontal lines of the grid. Thus, the accesses to the array elements are ideal for the CPU's caches.

B. Simulation Scenarios

The experimental data used in this paper is from the earth quake in Begkulu, Sumatra, on September 12, 2007. The dataset has a grid size of 2851×1801, leading to a memory usage of 233 MB. The simulated time equals 10 hours, requiring 7200 time steps, i.e. 5 seconds per time step. The experiments described in the paper were also conducted with three other datasets having different sizes and geographical data, but led to the same performance results.

III. Hardware Environment

The hardware used for tuning and analyzing the program is presented in Table I. The GPU cards have different CUDA compute capabilities and a different total number of cores whereas System A with a Tesla C1060 possesses only about the half of compute cores as System B having a Tesla C2075. The architectural difference between the GPUs is illustrated in Figure 2: The Tesla C2075 includes caches which were introduced with compute capability 2.0 respectively the Fermi hardware architecture. In detail, the C2075 possesses a 768 KB unified L2 cache that is shared by all streaming multiprocessors (SM) and individual L1 caches for the SMs of configurable size. In the presented experiments a cache size of 48 KB was used.

Property	System A	System B
GPU product name	Tesla C1060	Tesla C2075
HW Architecture	Tesla	Fermi
Compute Capability	1.3	2.0
Multi Processors (SM)	30	14
Cores per SM	8	32
Cores (total)	240	448
Global Memory	4096 MB	5375 MB (ECC)
Caches	none	L1 per SM, L2 for all SM
Driver version	304.88	310.32
CPU	Intel Xeon E5520	Intel Xeon E5-1603
Cores	4 Cores, 1 Socket	4 Cores, 1 Socket
Frequency	2.27 GHz	2.8 GHz
Main Memory	24 GB	8 GB

Table I. Specification of the experimental hardware environment.

On the software side, we used the CUDA 5.0 toolkit to compile the CUDA based version. The C++ compiler for the CPU code was `g++` from the GNU Compiler Collection version 4.6. For OpenACC, the PGI Compiler version 13.6 has been used. Although alternatives from the academic [10] as well as the industry supported open source community [11] are available, the commercial product was chosen as we expected increased quality of the compiled code from a vendor collaborating with the hardware manufacture.[1] Moreover, the OpenACC branch of the GCC seems still to be experimental and did not work in our setup.

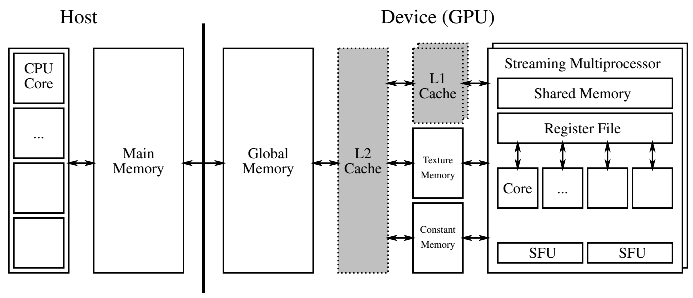


Figure 2. Overview of the memory hierarchy and GPUs used in this paper. Gray/dotted components are only available on the Fermi based Tesla C2075 device.

IV. CUDA Implementation

The sequential version of EasyWave was ported to the GPU using CUDA first. The implementation was incrementally optimized. After each optimization step, the improvement on the performance of the application was analyzed. In the following sections, the optimizations are discussed in detail.

[1] PGI was acquired by NVIDIA during the progress of the work presented in this paper.

A. Algorithmic Changes

1) Grid Point Update on GPU: According to the programming model of the GPU hardware, the sequential algorithm using nested loops (see Section II) had to be rewritten following the SIMD resp. SIMT paradigm. Thus, for the EasyWave port each GPU thread computes the physical properties of a single grid point. To ensure completion of the individual computational steps, computing the wave heights, fluxes, and the decision to expand the window is done in separated kernels.

This straightforward parallelization already delivered a substantial speedup of the application: The sequential version requires 348 seconds on the CPU of System A, respectively 305 seconds on System B's CPU. On the according GPUs the runtime is reduced to 162 seconds (Tesla C1060) and 28,4 seconds (Tesla C2075). This equals a speedup of 2.15 for the Tesla-based card, whereas the Fermi achieves a speedup of 10.7.

2) Parallel Window Extension: In the parallel version described above, the extension of the computational window is carried out by a single GPU thread. This can be parallelized as follows: The threads test in parallel if the threshold of the boundary cells residing in their computational window has been reached. For synchronization atomic instructions are used to store a boolean (non-zero) value that signals the extension of a window boundary. If an extension is necessary, this step is performed on the GPU by a single thread.

Compared to the GPU port of the main loop, the parallelized window extension reduces the runtime on the C1060 to 142 seconds, thus improving the performance by 13 %. The improvement is even more significant on the C2075 where the speed is enhanced by 46 % leading to a runtime of 15,3 seconds.

B. Hardware-specific Optimizations

Further performance improvements can be achieved by adapting the code to the architectural demands of the GPUs.

1) Memory Alignment: One of the most common tuning steps is to ensure the alignment of the memory used for the computation. Therefore, the memory used for the computation was allocated using `cudaMallocPitch`/`cudaMemcpy2D` to enable the hardware to optimize the memory accesses.

This modification resulted in a 4 % lower runtime for the C1060 card, and in negligible improvement for the C2075. The reason for this behaviour may be the additional computation for adjusting the boundaries of the computed window to the aligned memory addresses. Again, the use of CPU caches makes software improvements unnecessary.

2) Call by Value: A further enhancement was achieved by changing the way the arguments are passed to the kernel functions. Originally, the main array containing the pointers to the data arrays (see Section II) was passed directly to the GPU. When referencing an element in an data array, two accesses to the GPU memory where required: first, picking the element from the pointer array that contained the pointer to the data array. Then the required data element was accessed. This access pattern did not involve performance issues on the CPU as the cache would hold the values of the pointer array after an access. As the Tesla generation of Nvidia GPUs do not provide caches, the double memory access slows down the computation. Moreover, the parallel read on the pointer array is serialized by the GPU hardware and results in additional performance loss. This applies as well to variables that are constant during the execution of a kernel, like the current boundaries of the compute window.

To avoid these issues, all data arrays were passed directly to the kernel as pointer (in contrast to a pointer to a pointer array). Compared to the aligned memory optimization, the runtime of computation could be reduced by further 41% on the Tesla C1060. On the other hand, for the Fermi-based C2075 a relatively small improvement of about 6 % could be observed. As this card provides caches, the optimization does not come as much into effect as on the cache-less C1060.

3) Shared Memory: As the global memory containing the important computational data is the slowest memory type available on the GPU, avoiding accesses is likely to increase the speed of the computation. This is especially true for the stencil computations that are used by EasyWave and are memory-bound. Since the Tesla-generation cards do not have caches, using the Shared Memory of the Multiprocessors is often suggested as software-managed substitute. This common optimization technique was applied for both cards: Before running the computation, the area computed by a multiprocessor is loaded into its shared memory. When computation is finished on this scratch pad memory the computed values are stored back.

Running the kernel using shared memory significantly improves the performance on the Tesla-architecture card (C1060). Compared to the call by value version, the runtime is reduced to 55 %. In total, this leads to a runtime that is only about a fifth of the naïve port that transferred the core computation to the GPU (see Section IV-A). In contrast, when using shared memory on the Fermi-based C2075, the runtime increases compared to call by value version of the application. This can be accounted to the additional computational effort to copy data in and out of the shared memory emulating the cache's functionality, which is already present in the hardware.

C. Comparison

When comparing the performance of all optimization versions on both GPUs (see Figure 3), it is obvious that the optimizations done on the older Tesla C1060 card improve the performance significantly. Although, much effort and knowledge of the underlying hardware is required by the application programmer to achieve this speedup. Moreover, some well-known and often suggested tuning approaches, such as ensuring memory alignment, did not have the large impact as expected. Compared with changing the parameter passing method, the effort-benefit ratio of the latter seems to be better, nevertheless the gain in performance is surprising.

Comparing the performance of the algorithm between both cards, the most optimized version on the old-generation card gets very close to the performance of the Fermi-Card running the application with only the algorithm adjusted to the GPU's architecture (see Section IV-A). Concerning productivity, it is therefore reasonable to acquire hardware based on recent hardware architecture. Moreover, certain optimization techniques

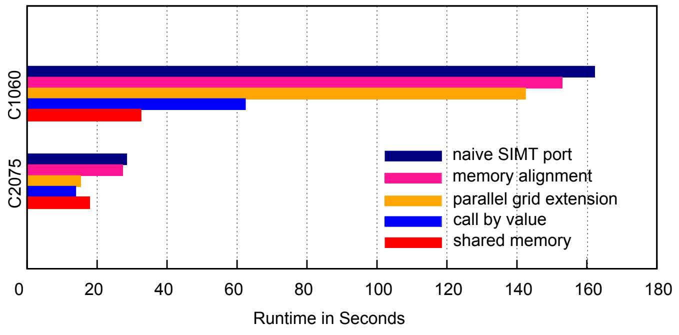


Figure 3. Runtime comparison of the discussed optimizations on both GPUs

leading to large performance gains on old hardware seem to provide low benefits on modern GPUs.

V. OpenACC Implementation

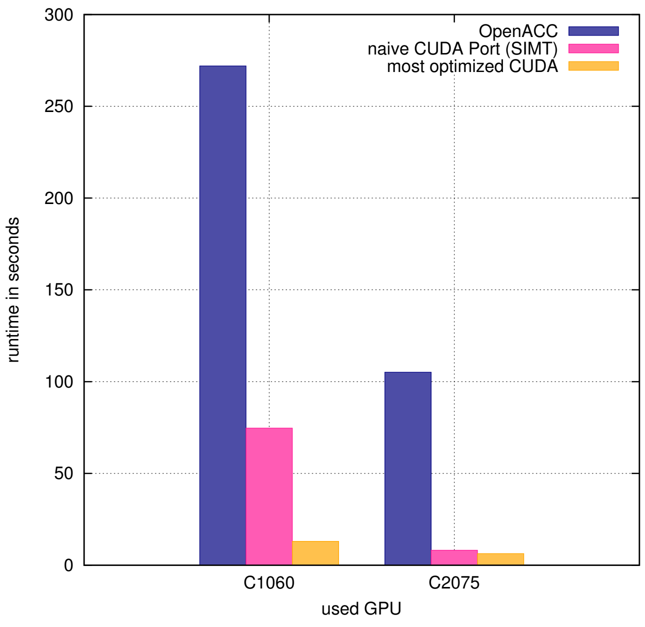
As intended by OpenACC, no changes in EasyWave's original sequential code, i.e. the loops performing the computation, were committed. OpenACC directives were added to ensure that data arrays reside in the GPU's memory and transfers between GPU and CPU are reduced to a minimum. To parallelize the code, the sequential `for`-loops were decorated with directives to convert the loops into kernels running on the GPU, i.e. `kernels` and `loop`. In total, this led to 21 additional lines of compiler directives compared to the sequential program having 462 lines of code. In comparison, 248 additional lines were required for the most tuned CUDA version of the application.

The OpenACC code was compiled for the different compute capabilities of the two GPUs using the PGI compiler version 13.6. On the Tesla C1060 a disappointing speedup of 1.15 compared to the sequential CPU version was achieved. Similar, on the system containing the Tesla C2075 the speedup was at 2.67 which is also much less than the native CUDA version discussed in Section IV-A1 achieved (speedup of 2.15 resp. 10.7).

A deeper analysis revealed that the parallelized loops exhibit different performance compared to the native and optimized CUDA version. The update of the wave height, which uses a TLC-stencil (see Section II) performs worse compared to both CUDA versions on both GPUs. In contrast, the OpenACC version of the flux update, that uses a BRC-stencil, outperforms the optimized CUDA code on the Tesla C2075, whereas it reaches similar performance on the C1060 as shown in Figure 4. We assume this as a compiler issue as the structure of of the functions is very similar and only differs in the memory access pattern. This problem has already been discussed with the vendor's support, but is still in discussion at the time of writing.

VI. Conclusion

This paper presents performance results of different parallel versions of the Tsunami simulation EasyWave. While the native CUDA version achieves a speedup of 10.7 on a Tesla C2075, the speedup of the OpenACC version was only 2.67 compared the sequential CPU version.



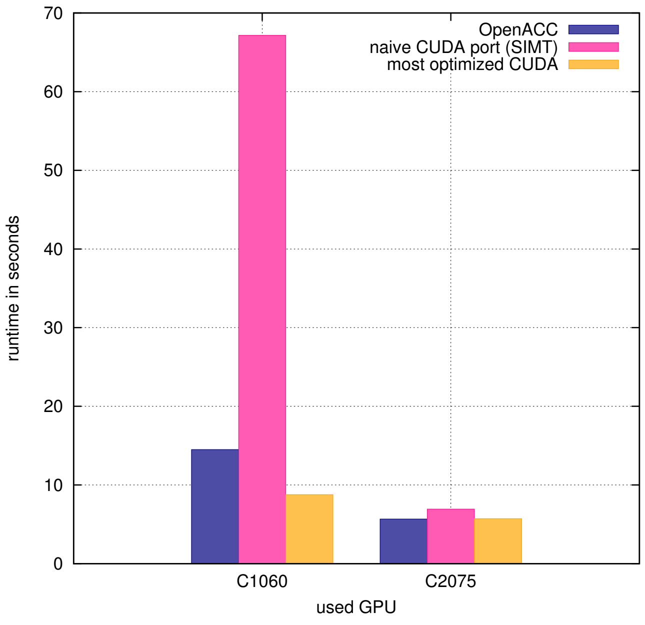


Figure 4. Performance of OpenACC compiled code for the loops performing the wave height update using TLC-stencil (top) and the flux update using BRC-stencil (bottom).

Further, it was shown that recent hardware has a positive effect on the computational speed when a program is directly programmed for CUDA. This is especially true if the source code is not tuned manually to exploit the features of the hardware but is only programmed following the SIMD paradigm of GPUs. In contrast, it is necessary for a programmer to be much more aware of hardware details to gain a significant performance improvement on old hardware. Thus, the usage of recent hardware can unburden a non-expert programmer from the task of tuning their specific application to the hardware.

Even more, from a (scientific) application programmer's perspective, the usage of Open-ACC seems to be promising as very few code changes are required to enable support for accelerators. It was further shown that compute kernels differing only in the memory access pattern can result in very different performance when using OpenACC. This emphasizes the crucial role of compiler support for the OpenACC standard.

REFERENCES

[1] J. Wächter, A. Babeyko, J. Fleischer, R. Häner, M. Hammitzsch, A. Kloth, and M. Lendholt, "Development of tsunami early warning systems and future challenges," *Natural Hazards and Earth System Science*, vol. 12, no. 6, pp. 1923–1935, 2012. [Online]. Available: http://www.nat-hazards-earth-syst-sci.net/12/1923/2012/

[2] M. Hammitzsch, F. J. Carrilho, O. Necmioglu, M. Lendholt, S. Reißland, J. Schulz, R. Omira, M. Comoglu, N. M. Ozel, and J. Wächter, "Meeting unesco-ioc icg/neamtws requirements and beyond with tridec's crisis management demonstrator for tsunamis," in *23rd International Ocean and Polar Engineering Conference - ISOPE*, Anchorage, USA, 2013.

[3] A. Babeyko, online, https://media.gfz-potsdam.de/gfz/wv/05_Medien_Kommunikation/Bildarchiv/Erdbeben_Japan/temp_xs/33408642-110313_Tsunami_Japan_2_DRUCK.png.

[4] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *Micro, IEEE*, vol. 28, no. 2, pp. 39–55, 2008.

[5] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science and Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[6] openacc.org, *The OpenACC Application Programming Interface*, 2011, version 1.0, http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf. [Online]. Available: http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf

[7] S. Wienke, P. Springer, C. Terboven, and D. Mey, "OpenACC — First Experiences with Real-World Applications," in *Euro-Par 2012 Parallel Processing*, ser. Lecture Notes in Computer Science, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds. Springer Berlin Heidelberg, 2012, vol. 7484, pp. 859–870. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32820-6_85

[8] D. J. Greenslade, A. Annunziato, A. Y. Babeyko, D. R. Burbidge, E. Ellguth, N. Horspool, T. S. Kumar, C. P. Kumar, C. W. Moore, N. Rakowsky, T. Riedlinger, A. Ruangrassamee, P. Srivihok, and V. V. Titov, "An assessment of the diversity in scenario-based tsunami forecasts for the indian ocean," *Continental Shelf Research*, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0278434313002021

[9] A. Babeyko, "EasyWave: fast tsunami simulation tool for early warning," February 2012, ftp://ftp.gfz-potsdam.de/pub/home/mod/babeyko/easyWave/easyWave_About.pdf.

[10] R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. Sande, "accULL: An OpenACC Implementation with CUDA and OpenCL Support," in *Euro-Par 2012 Parallel Processing*, ser. Lecture Notes in Computer Science, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds. Springer Berlin Heidelberg, 2012, vol. 7484, pp. 871–882. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32820-6_86

[11] E. Gavrin, "Openacc branch [openacc-1_0-branch]," Gnu GCC Mailing list, online http://gcc.gnu.org/ml/gcc/2013-09/msg00235.html, Sep 2013. [Online]. Available: http://gcc.gnu.org/ml/gcc/2013-09/msg00235.html