Strategies to Improve the Performance and Energy Efficiency of Stencil Computations for NVIDIA GPUs

Pablo José Pavan[1], Matheus da Silva Serpa[1], Víctor Martínez[1], Edson Luiz Padoin[1,2], Jairo Panetta[3], Philippe O. A. Navaux[1]

[1] Informatics Institute - UFRGS - Porto Alegre - RS - Brazil

{pablo.pavan,msserpa,victor.martinez,navaux}@inf.ufrgs.br

[2]Department of Exact Sciences and Engineering - UNIJUI - Ijuí - RS - Brazil

padoin@unijui.edu.br

[3]Computer Science Division ITA - São José dos Campos – SP – Brazil

jairo.panetta@gmail.com

Abstract. *Energy and performance of parallel systems are an increasing concern for new large-scale systems. Research has been developed in response to this challenge aiming the manufacture of more energy efficient systems. In this context, we improved the performance and achieved energy efficiency by the development of three different strategies which use the GPU memory subsystem (global-, shared-, and read-only- memory). We also develop two optimizations to use data locality and use of registers of GPU architecture. Our developed optimizations were applied to GPU algorithms for stencil applications achieve a performance improvement of up to* 201.5% *in* $K80$ *and* 264.6% *in* $P100$ *when used shared memory and read-only cache respectively over the naive version. The computational results have shown that the combination of use read-only memory, the Z-axis internalization of stencil application and reuse of specific architecture registers allow increasing the energy efficiency of up to* 255.6% *in* $K80$ *and* 314.8% *in* $P100$*.*

1. Introduction

Different applications in areas such as computational physics, weather forecast, oil exploration, climate modeling and atomic simulation require high performance with acceptable power consumption. These scientific applications use stencil computations that include both implicit and explicit partial differential equations (PDE) solvers [Datta et al. 2008]. Besides the stencils scientific importance, they are interesting as a performance and energy consumption evaluation benchmark because they have abundant parallelism and low computational intensity, offering opportunities for on-chip parallelism and challenges for associated memory systems [Datta et al. 2008].

Scientific simulations may consume weeks and most of these time is spent in stencil computations [de la Cruz and Araya-Polo 2011]. Continuous changes in the microprocessors fabrication process have increased the performance of its products and influenced state-of-the-art HPC systems. However, this exponential increase in performance also leads to an exponential growth in power demand [Laros et al. 2009, Dong et al. 2010, Padoin et al. 2013a]. Reductions in the applications execution time are also relevant for energy consumption because energy is saved when hardware resources are used for a shorter time.

Memory performance is particularly important for stencil computations once they are typically memory-bound. For this reason, optimize the memory accesses is one of the keys to reducing the performance and energy consumption. In this paper, we improved the performance and energy efficiency of stencil applications by better use of the memory subsystem. We focus on analyzing the use of different Graphics Processing Units (GPU) memories aiming to improve performance, energy consumption, and energy efficiency. The main contributions of this paper are:

- propose of three strategies which use different GPU memories (*global-*, *shared-*, and *read-only-* memory);
- extend these strategies by use data locality and registers optimization;
- performance evaluation of these optimizations on Tesla K80 and Tesla P100 GPUs.

The remaining sections of this paper are organized as follows. Section 2 discusses the related work. In Section 3 we present the stencils application, the details of the optimizations we developed and the evaluation methodology. Also, in Section 4 we address the results obtained from the experiments. Finally, the Section 5 emphasizes the scientific contribution of the work and notes several challenges that we can address in the future.

2. Related Work

Several studies evaluated the stencil performance and energy efficiency in CPUs and GPUs. Despite that, processors and accelerators remain as the component with the highest power demand of the systems [Feng et al. 2005]. GPUs are made aiming massively parallel processing and to achieve this they use hundreds of processing units working together. These characteristics lead to its superior energy efficiency when compared with CPUs [Padoin et al. 2013b].

Micikevicius *et al.* [Micikevicius 2009] compared the performance of a stencil ported from CPU to GPU. Their version of the stencil running in a GPU achieved an order of magnitude higher than running in a contemporary CPU. They conclude that is possible to improve their results by the usage of shared memory to reduce communication overhead. Bauer *et al.* [Bauer et al. 2011] showed that the main bottleneck in GPU applications is related to the memory subsystem. To reduce its impact, they used DMA warps to improve memory transfer between on-chip and off-chip memories. They achieved a speedup up to 3.2$\times$ on several kernels form scientific applications.

Schäffer and Fey [Schäfer and Fey 2011] evaluate a set of algorithms on Fermi GPUs. They evaluate micro-benchmarks using shared memory and found that using only the L1 cache creates a problem for its limited throughput. Also, the L2 cache is not a good option because of cache blocking. They conclude that a new alternative to use shared memory was needed to overcome communication bottleneck. Falch and Elster [Falch and Elster 2014] proposed the usage of a manually managed cache to combine the memory from multiple threads. Using their technique, they achieved a speedup of up to 2.04 in a synthetic stencil. They concluded that manual caching is an effective approach to improve memory access and that applications with regular access patterns are suitable to implement their technique.

Zhou *et al.* [Zhou et al. 2016] points that the use of GPUs enables considerable gains in performance compared to using CPU. They have applied GPUs successfully in many computations and memory intensive realms due to its superior performances in the float-pointing calculation, memory bandwidth, and power consumption. The results obtained show a speedup of up to 50 times using GPU algorithm rather than CPU algorithm. In

similar works, Zhou *et al.* [Zhou et al. 2012] obtained a speedup between 10 and 15 times using a GPU rather than CPU.

Xue *et al.* [Xue et al. 2015] also make comparisons between GPU and CPU implementation. They obtained a speedup up to 18 times in the GPU-based implementation of a time-reversal imaging micro-seismic event location. Also, Nikitin *et al.* [Nikitin et al. 2012] obtained average speedup up to 46 times using GPU for compared to CPU for processing a synthetic seismic data set (data compression, de-noising, and interpolation).

Maruyama and Aoky [Maruyama and Aoki 2014] present a method for stencil computations on the NVIDIA Kepler architecture that uses shared memory for better data locality combined with warp specialization for higher instruction throughput, their method achieves approximately 80% of the value from roof line model estimation. Hamilton *et al.* [Hamilton et al. 2015] investigate the computational performance of GPU-based stencil operations using stencils of varying shape and size (ranging from seven to more than 450 points in size). They found that using an NVIDIA K20 GPU, data movement, rather than computing, was the bottleneck, and as such, the performance obtained can be attributed to the effects of the L2 and texture caches on the card.

Nasciutti and Panetta [Nasciutti and Panetta 2016] did a performance analysis of 3D stencils on GPUs focusing on the proper use of the memory hierarchy. They conclude that the preferred code is the combination of read-only cache reuse, inserting the Z loop into the kernel and register reuse. Different to other approaches that allocate workload on CPU and GPU architectures, or works that use GPUs to achieve considerable performance gains when compared to traditional CPU architecture. Our goal is to increase performance and energy efficiency of stencil application applying methods and optimization to use different memory levels of the GPUs.

3. Evaluation Methodology

In this section, we show the stencil model we use as a case study. We also present the GPUs architectures, the optimizations evaluated and the experimental methodology.

3.1. Stencil Computations

The computational performance of GPU-based stencils has great scientific importance as it is used in many areas of scientific computing. Stencil computations are present from a simple Jacobi iterations until extremely complex solutions of non-linear Partial Differential Equations (PDE) [Dubey 2014]. A stencil application calculates the value of one point of the grid in the current iteration using the value of this same point and its neighbors in the previous iteration [Datta et al. 2008].

3.1.1. Fletcher Model in Isotropic Acoustic Wave Propagation

The modeling simulates the collection of data in a seismic survey, as in Figure 1. At intervals of, equipment coupled to the ship emits waves that reflect and refract on changes of the medium in the subsoil. Eventually, these waves return to the surface of the sea, being collected by specific microphones (geophones) coupled to cables towed by the ship. The set of signals received by each geophone over time constitutes a seismic trace. For each wave emission, the seismic traces of all cable geophones are recorded. The ship continues to sailing and emits signals over time.

Acoustic wave propagation approximation is the current backbone for seismic imaging tools. It has been extensively applied to imaging potential oil and gas reservoirs beneath

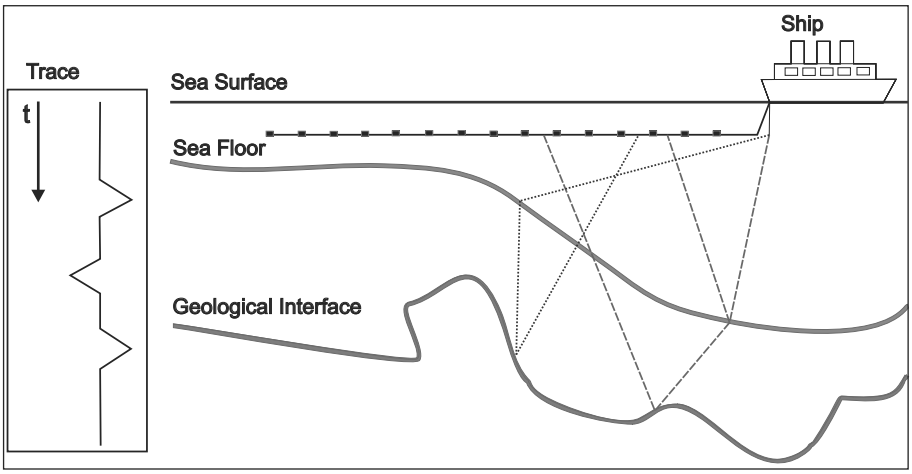


Figure 1: Data collection (traces) in maritime seismic survey.

salt domes. We consider the model formulated by the isotropic acoustic wave propagation under Dirichlet boundary conditions over a finite 3D rectangular domain, prescribing to all boundaries, and the isotropic acoustic wave propagation. Propagation speed depends on variable density, the acoustic pressure, and the media density. The Numerical method solves Equation 1 and is detailed in [Vilela 2017]. Figure 2 illustrates the acoustic wave propagation stencil.

$$\begin{aligned} C_{i,j,k} =& a_0 C_{i,j,k} \\ &+ a_1(C_{i-1,j,k} + C_{i+1,j,k} + C_{i,j-1,k} + C_{i,j+1,k} + C_{i,j,k-1} + C_{i,j,k+1}) \\ &+ a_2(C_{i-2,j,k} + C_{i+2,j,k} + C_{i,j-2,k} + C_{i,j+2,k} + C_{i,j,k-2} + C_{i,j,k+2}) \\ &+ a_3(C_{i-3,j,k} + C_{i+3,j,k} + C_{i,j-3,k} + C_{i,j+3,k} + C_{i,j,k-3} + C_{i,j,k+3}) \end{aligned} \tag{1}$$

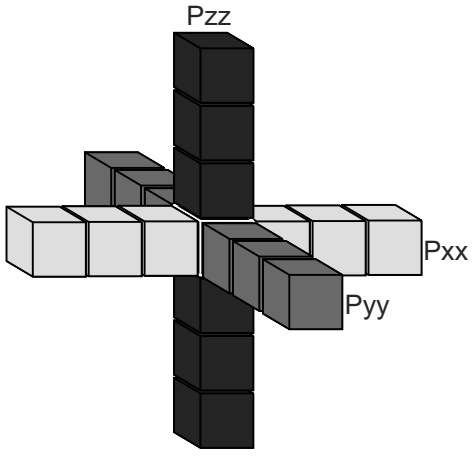


Figure 2: Acoustic Wave Propagation 7 point Stencil.

3.2. Optimization Strategies

Current GPU architectures provide memories with different characteristics compared with CPUs. One of the main differences between GPUs and CPUs is the way their memory

subsystem work. In a CPU, access to memory is done by obtaining their data from caches. Usually looking on L1, L2, L3, and DRAM in that order. On the other hand, in a GPU the L1 memory cache, is used specifically for accesses to the stack and register spill, i.e., when too many local variables do not fit in the register file, and thus some of it has to be cached. L2 memory is used for global accesses requested by stream processors. There are also registers files, a shared memory, a texture memory and a global memory with different characteristics such as size, speed, read-only memory and in the way that is possible to use them. These registers were not available in NVIDIA GPUs before Kepler architecture. In Figure 3, we show an overview of the Kepler GPU architecture.

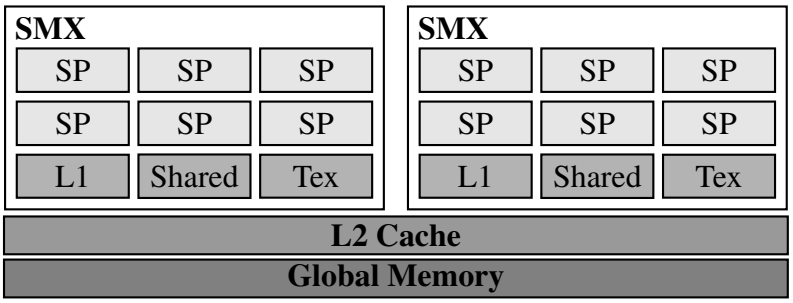


Figure 3: Sample of the memory subsystem on a NVIDIA Kepler architecture

To exploit the use of different memory levels available on current GPU, we develop three versions of a stencil kernel using each one of the GPU memories. Each stencil version, give us a different insight of the performance and capabilities of the GPU memory subsystem.

- The first version called *naive* take no advantage of any of the GPU high-speed memories and access data only from global memory.
- The second version called *shared* stores one part of the stencil data in the shared memory scratchpad. The shared-memory version also uses the GPU resources that the naive version uses, the main difference is that this version also uses the shared-memory available on each SMX (Streaming Multiprocessors). Each one of the SMX has one internal shared-memory to store data as shown in Figure 3. In this version, data is manually allocated by the programmer through the use of the *shared* directive, indicating such data will be shared among all the GPU threads. The compiler automatically configures the space division between the L1 cache memory and the shared cache memory, choosing one of three options: 16 KB for the L1 cache and 48 KB for the shared cache, 32 KB for each, or 48 KB for the L1 cache and 16 KB for the shared cache.
- The third version called *read-only* stores most read data in a read-only texture memory which is faster than shared memory but works with read-only data. This version takes advantage of the read-only cache which is the SMX memory bank that stores only read data, it is also called texture memory. Originally it was used only for textures, but starting with the Kepler architecture any data can be stored in this cache by using the C-99 directive `const restrict`. The programmer may also explicitly use this cache through the intrinsic `lgd()`.

We developed two optimizations for each of the versions to evaluate improvements in performance and energy efficiency by reusing the Z direction data. Reusing Z direction data is named *internalization*.

- The *int.z* version takes advantage of data locality by storing stencil data for direction Z. This optimization consists of the internalization of the Z-axis into the threads. Doing the internalization ensures that neighboring Z-blocks execute sequentially, increasing the reuse of L2 cache data. Direction Z data is used to calculate subsequent points in the X-Y direction.
- The *int.z.reg* version consists of combining the *int.z* with the usage of registers to store the Z direction points. For example, to calculate the point Z3 in a 13 points stencil, the neighboring points in X and Y, as well as points Z1, Z2, Z3, Z4 and Z5 are required. In order to calculate the points in Z4, points Z2, Z3, Z4 and Z5 would be availed, and it is necessary to request the global memory only points Z6, as well as the neighbors in X and Y.

3.3. Experimental Methodology

Our experiments were developed in two NVIDIA GPU card. The first is an NVIDIA K80 GPU. This card is a Kepler architecture GPU with 2496 CUDA cores. The second card is an NVIDIA P100 GPU. This card is a Pascal architecture GPU with 3584 CUDA cores. Table 1 describes in detail the environments we used.

Device	Tesla $K80$	Tesla $P100$
CUDA Cores	2496	3584
Registers	13 x 512kB	56 x 256KB
Memory	13 x 128KB L1 / *shared*	56 x 64KB *shared*
	1536KB L2	4096KB L2
	13 x 48KB *texture (read-only)*	56 x 24KB L1 / *texture (read-only)*
	12GB GDDR5 *global memory*	16GB GDDR5 *global memory*

Table 1: Configuration of GPU system.

We used NVIDIA Management Library (NVML) to measure the power usage. Regarding the energy efficiency, we used performance divided by average power. Each experiment was executed 20 times, we show average values as well a 95% confidence interval calculated with Student's t-distribution.

4. Results

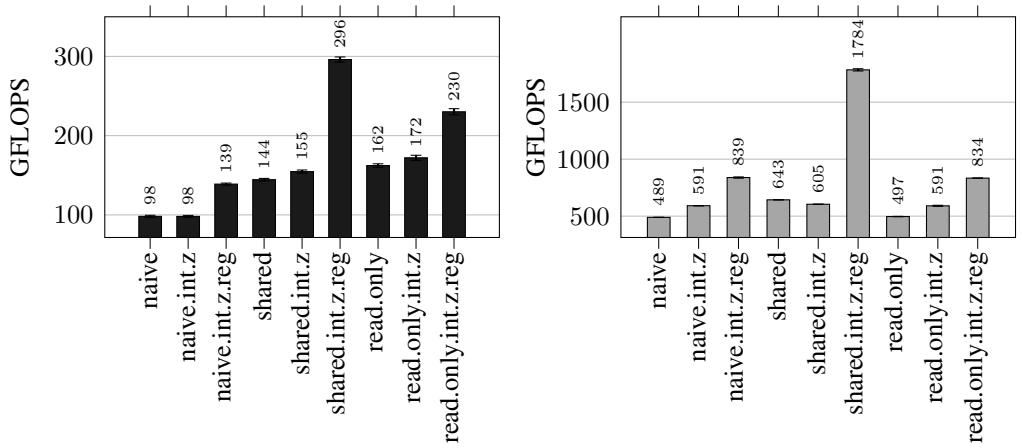
This section shows the optimizations techniques we used to improve the performance and energy efficiency of a stencil application. The stencil we used simulates the propagation of a single wavelet over time. To create the simulation, it solves the isotropic acoustic wave propagation with constant density under Dirichlet boundary conditions over a 3D domain. The stencil is a 31-arm with a $(1024 \times 256 \times 256)$ input size.

In the following subsections, we describe each optimization and analyze how they address the performance and energy efficiency improvements. We also show the results obtained by using the three different memories and the results of the optimizations applied in each of them, on an NVIDIA Kepler architecture and NVIDIA Pascal architecture.

4.1. Performance Improvements

In this subsection, we show the Performance improvements obtained by using different GPU memory subsystem (*global-* , *shared-* , and *read-only*-memory) and two optimization techniques over a stencil kernel. In Figure 4 is showed the performance achieved for

each kernel version and optimizations performance. $P100$ features $5x$ more performance that $K80$ GPU for this stencil kernel using Naive version. The performance achieved in $K80$ was 98.1 GFLOPS while in $P100$ was 489.3 GFLOPS.



(a) K80 (b) P100

Figure 4: Performance gains over $K80$ and $P100$

Applying the first optimization technique *int.z* which stores data from direction Z in local variables to take advantage of the data locality by reusing these data in the subsequent iterations, performance gains were not realized in $K80$ (Figure 4(a)). Otherwise, in P100 (Figure4(b)), the performance were improved by up to $20,8\%$ using the this *int.z* technique over a *naive* version. When the second optimization *int.z.reg* was employed, which consists of the *int.z* optimization along with the use of the register file to store the Z points, the performance was increased from 98.1 GFLOPS to 138.8 GFLOPS in $K80$, from 489.3 GFLOPS to 838.9 GFLOPS in $P100$, which represent an increase of 41.5% and 71.4% respectively.

Although the performance was improved using optimizations techniques, the *naive* version does not take advantage of fast GPU memories as *shared* memory. Thus, we improved the *naive* version by used the *shared* memory scratch pad to store a slice of data that is reused by the threads of the same block.

To the second version, the performance was increased from 98.1 GFLOPS to 144.4 GFLOPS in $K80$, from 489.3 GFLOPS to 643.2 GFLOPS in $P100$. In this version the data were manually allocated using the *shared* directive, indicating a piece of data shared among all threads. The performance improvement using the shared memory compared to the naive was 47.10% on the K80 and 31.45% on the P100. In the P100, the performance is lower because in Pascal architecture the number of SMs were increased and the number of SPs in each SM were decreased. In this case, each P100 SM has less shared memory than a K80 SMX.

Similarly to the first version, we also applied the *int.z* and *int.z.reg* optimizations in this version aiming to improve the performance of the memory operations. The *int.z* optimization over stencil represented an improvement of 7.07% in $K80$ performance, which achieve 154.6 GFLOPS. Differently, in $P100$ the performance was reduced when used this optimization.

The second optimization (*int.z.reg*) over *shared* memory performed in $K80$ increased the performance in 104.9% when compared to *shared* memory version without optimization and 201.5% when compared to *naive* without optimizations. Using this optimization

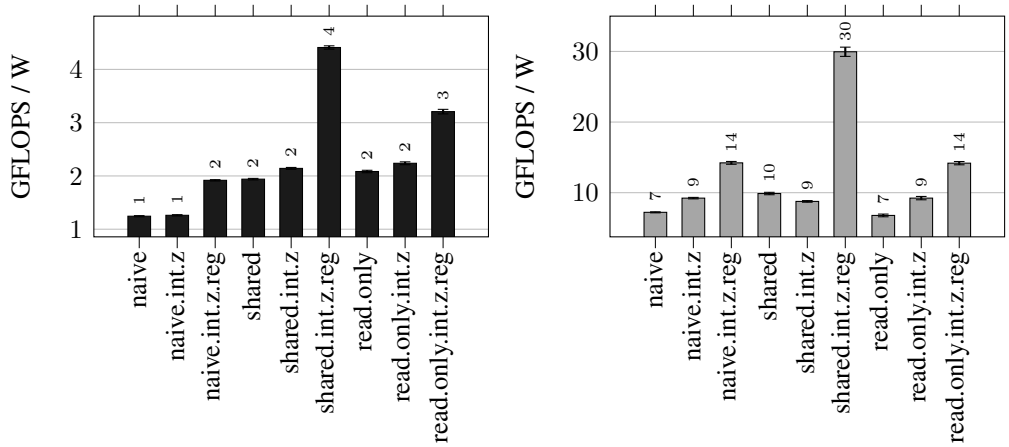
and *shared* Memory version in a stencil application was achieved the best performance, 296.04 GFLOPS. The gains were greater when the version with the second optimization was applied in $P100$. The performance increased in 177.3% when compared to *shared* memory version without optimization. Since the data are stored in the *shared* memory they are not updated. So, we may take more advantage if the *read-only* memory is used. The *read-only* memory is faster than *shared* memory but exclusively used for read-only operations.

Running the *read.only* version, which define that *global* memory reads are stored in the *read-only* memory using the *lgd()* intrinsic, the performance achieved 162.2 GFLOP in the $K80$ 496.5 GFLOPS in the $P100$. The performance improvement of this strategy compared to the naive on K80 was 65.26% and 1.48% on P100. The low improvement occurs on P100 architecture because on Pascal, the read-only memory is shared with the L1 and the compiler defines the distribution for each memory. In Kepler, the read- only memory is not shared with any other memory.

Adding the *int.z* optimizations over this third version of the kernel the performance increase 6.03% in the $K80$ and 18.9% in the $P100$. The performance is also increased when used the *int.z.reg* optimization over stencil performing in the *read-only* memory. Implementing the *int.z.reg* that also uses the register file, the performance was improved by up to 41.8% in the $K80$ and 67.9% in the $P100$.

4.2. Energy Efficiency Improvements

In this subsection, we show the energy efficiency improvements obtained by using different GPU memory subsystem (*global-* , *shared-* , and *read-only*-memory) and two optimization techniques over a stencil kernel. Figure 5 showed the energy efficiency achieved for each kernel version and optimizations performance.



(a) K80 (b) P100

Figure 5: Energy Efficiency gains over $K80$ and $P100$

The gains in energy efficiency are similar to those achieved with the increase in performance. The energy efficiency of the $P100$ (Figure 5(a)) was 5.8 times greater than $K80$ (Figure 5(b)) to *naive* version. The first optimization technique *int.z* overtakes the normal version and also improvement the energy efficiency of up to 28.8% compared with the *naive* version in $P100$. When the second optimization *int.z.reg* was employed, the energy efficiency increase 54.8% in the $K80$ and 96.9% in the $P100$.

Analysing the energy efficiency gains to the second version of the kernel, it was 56.4% in the $K80$ and 36.8% in the $P100$ over Naive version without optimization. Applying the *int.z* and *int.z.reg* optimizations in this version the energy efficiency improvement in $K80$ and reduces in $P100$. However, the main gains were achieved with the second optimization (*int.z.reg*) over *shared* memory. It increases the energy efficiency in 255.6% in the $K80$ and 314.8% in the $P100$ when compared to the naive version without optimizations. The best efficiency was achieved when the *int.z.reg* optimization was used over *read-only* version. In $K80$ it achieved 3.21 GFLOP/W.

5. Conclusion

Nowadays, the use of GPUs in HPC systems has become a popular choice among the top-ranked platforms. However, to achieve even greater performance and energy efficiency is necessary to exploit the different memory levels available. Several scientific applications make use of stencil computations to their model simulations and are performed in GPU platforms. Stencils have both implicit and explicit partial differential equations (PDE) being interesting as an architectural evaluation benchmark. Their computing present low computational intensity, once that these applications are typically memory-bound. In this form, memory optimizations are important for to use the fastest memories available in GPUs and increase their energy efficiency.

In this paper, we improved the performance and achieved energy efficiency by the development of three different strategies which use the GPU memory subsystem (*global*, *shared*, and *read-only* memory). We also develop two optimizations to use data locality and use of registers of GPU architecture. To analyze the impact of our proposed strategies and optimizations, we applied them to a stencil application and ran on Tesla K80 and Tesla P100 GPUs. Our developed strategies to GPU algorithms achieved the performance improvement of up to 201.5% in $K80$ and 264.6% in $P100$ when were used *shared* memory and *read-only* cache respectively over the *naive* version. These increases in computational performance also improve the energy efficiency. The main gains were achieved with (*int.z.reg*) optimization over *shared* memory which increases the energy efficiency in 255.6% in $K80$ and 314.8% in $P100$ when compared to the naive version without optimizations.

Changes in the GPU architecture, as in the case of the introduction of the *read-only* cache in the Kepler architecture, can generate changes in the results presented in this work. In the future, we plan to investigate methods and optimizations aiming to achieve gains in stencil applications over new NVIDIA architecture and Intel Xeon Phi.

Acknowledgment

This research received funding from the EU H2020 Programme and from MCTI/RNP-Brazil under the HPC4E project, grant n.º 689772. It was also supported by Intel under the Modern Code project, and Petrobras 2016/00133-9.

References

Bauer, M., Cook, H., and Khailany, B. (2011). Cudadma: Optimizing gpu memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 12:1–12:11, New York, NY, USA. ACM.

Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., and Yelick, K. (2008). Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 4. IEEE Press.

de la Cruz, R. and Araya-Polo, M. (2011). Towards a multi-level cache performance model for 3d stencil computation. *Procedia Computer Science*, 4:2146–2155.

Dong, Y., Chen, J., and Tang, T. (2010). Power measurements and analyses of massive object storage system. In *Proceedings of CIT*, pages 1317–1322. International Conference on Computer and Information Technology (CIT), IEEE Computer Society.

Dubey, A. (2014). Stencils in scientific computations. In *Proceedings of the Second Workshop on Optimizing Stencil Computations*, pages 57–57. ACM.

Falch, T. L. and Elster, A. C. (2014). Register caching for stencil computations on gpus. In *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 479–486. IEEE.

Feng, X., Ge, R., and Cameron, K. W. (2005). Power and energy profiling of scientific applications on distributed systems. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 34–34. International Conference on Performance Engineering, IEEE.

Hamilton, B., Webb, C. J., Gray, A., and Bilbao, S. (2015). Large stencil operations for gpu-based 3-d acoustics simulations. *Proc. Digital Audio Effects (DAFx),(Trondheim, Norway)*.

Laros, J., Pedretti, K., Kelly, S., VanDyke, J., Ferreira, K., Vaughan, C., and Swan, M. (2009). Topics on measuring real power usage on high performance computing platforms. In *Proceedings...*, pages 1–8. International Conference on Cluster Computing and Workshops (ICCC).

Maruyama, N. and Aoki, T. (2014). Optimizing stencil computations for nvidia kepler gpus. In *Proceedings of the 1st International Workshop on High-Performance Stencil Computations, Vienna*, pages 89–95.

Micikevicius, P. (2009). 3d finite difference computation on gpus using cuda. In *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84, New York, NY, USA. ACM.

Nasciutti, T. C. and Panetta, J. (2016). Impacto da arquitetura de memória de gpgpus na velocidade de computação de estênceis. In *XVII Simpósio de Sistemas Computacionais (WSCAD-SSC)*, pages 1–8, Aracaju, SE.

Nikitin, V. V., Duchkov, A. A., and Andersson, F. (2012). Parallel algorithm of 3d wavepacket decomposition of seismic data: Implementation and optimization for gpu. *Journal of Computational Science*, 3(6):469–473.

Padoin, E. L., de Oliveira, D. A. G., Velho, P., Navaux, P. O. A., and Mehaut, J.-F. (2013a). ARM-based cluster: Performance, Scalability and Energy Efficiency. In *4th Workshop on Applications for Multi-Core Architectures (WAMCA SBAC-PAD)*, pages 1–6, Porto de Galinhas, PB, Brasil.

Padoin, E. L., Pilla, L. L., Boito, F. Z., Kassick, R. V., Velho, P., and Navaux, P. O. (2013b). Evaluating application performance and energy consumption on hybrid cpu+ gpu architecture. *Cluster Computing*, 16(3):511–525.

Schäfer, A. and Fey, D. (2011). High performance stencil code algorithms for gpgpus. *Procedia Computer Science*, 4:2027 – 2036. Proceedings of the International Conference on Computational Science, ICCS 2011.

Vilela, R. F. (2017). Perfilagem do problema de resolução da equação da onda por diferenças finitas em coprocessador xeon phi.

Xue, Q., Wang, Y., Zhan, Y., and Chang, X. (2015). An efficient gpu implementation for locating micro-seismic sources using 3d elastic wave time-reversal imaging. *Computers & Geosciences*, 82:89–97.

Zhou, G., Zhang, X., Lang, Y., Bo, R., Jia, Y., Lin, J., and Feng, Y. (2016). A novel gpu-accelerated strategy for contingency screening of static security analysis. *International Journal of Electrical Power & Energy Systems*, 83:33–39.

Zhou, J., Unat, D., Choi, D. J., Guest, C. C., and Cui, Y. (2012). Hands-on performance tuning of 3d finite difference earthquake simulation on gpu fermi chipset. *Procedia Computer Science*, 9:976–985.