Transformation of Scientific Algorithms to Parallel Computing Code: Single GPU and MPI multi GPU Backends with Subdomain Support

Björn Meyer, Christian Plessl
Department of Computer Science
University of Paderborn, Germany
{bjoern.meyer, christian.plessl}@uni-paderborn.de

Jens Förstner
Department of Physics
University of Paderborn, Germany
jens.foerstner@uni-paderborn.de

***Abstract*—We propose an approach for high-performance scientific computing that separates the description of algorithms from the generation of code for parallel hardware architectures like Multi-Core CPUs, GPUs or FPGAs. This way, a scientist can focus on his domain of expertise by describing his algorithms generically without the need to have knowledge of specific hardware architectures, programming languages, APIs or tool flows. We present our prototype implementation that allows for transforming generic descriptions of algorithms with intensive array-type data access to highly optimized code for GPU and multi GPU cluster systems. We evaluate the approach for an example from the domain of computational nanophotonics and show that our current tool flow is able to generate efficient code that achieves speedups of up to $15.3\times$ for a single GPU and even $35.9\times$ for a multi GPU setup compared to a reference CPU implementation.**

***Keywords*-scientific computing; code generation; stencil computation; graphics processing unit;**

I. Introduction and Approach

In many research areas, the dynamic behaviour of complex (physical, sociological, economical) systems is simulated by numercial computer models with sufficient accuracy. But even on the latest traditional computer hardware architectures, the implied computational effort is beyond the hardware capabilities. For example, the physical simulation of simple nanophotonic structures can take multiple days to gain converged results [1].

In recent years, various novel parallel hardware architectures, like Multi-Core CPUs, Symetric Multi-Processing machines (SMP), Graphic Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs) have been introduced to the scientific computing market. They enable researchers to gain significant speedups for a certain class of applications like the mentioned simulation of nanophotonic structures [2]. This raises the question, why only a small portion of the available scientific software packages take an advantage of those novel hardware architectures, although it is expected that most of them could benefit from highly parallel architectures. We identified two main obstacles that lead to this situation: the required shift to a dedicated parallel programming paradigm and the implied extra effort for getting familiar with architecture details.

The intention of our approach is to meet those obstacles by *separating the algorithm description*, that shall be captured by a domain expert, *from the mapping to a particular hardware architecture*, which is performed by an *automated code generation and optimization tool flow*. Using this technique, domain experts will be able to formulate problems in a high-level language, which is agnostic of the target hardware architecture. One important similarity, the considered algorithms share with each other, is the regular memory access pattern, which is used to update a position in space for a defined domain. We expect that knowing the particular memory access pattern put us into the position to generate even higher performing code using benchmark-driven optimization.

Especially in the context of auto-tuning, several promising ideas and frameworks have been presented in the literature in recent years [3]. Most provide a hardware-agnostic algorithm description and a code generation facility with benchmark-steered optimization, however, to our knowledge only our approach combines those features with a powerful concept of subdomains and an intermediate language representation, that allowed us to support generation of single GPU as well as multi GPU (via Message Passing Interface (MPI)) code with minimal development effort.

The effectiveness of several implemented key concepts is shown for the widely used FDTD algorithm which solves the Maxwell PDEs and allows for simulation of electrodynamical field propagation. We show for specific simulation parameters how our implementation of the transformation chain can produce not only high-quality runnable code, but more importantly, how an automated optimization can be achieved without the need to change the description of the algorithm. We evaluate our results for a single GPU and a multi GPU implementation.

II. Implementation of our approach

We accomplish the mentioned *separation of algorithm description* from the *maping to hardware* by introducing a domain-specific language (DSL) in close collaboration with domain experts. Hence, our DSL is uncluttered and expressive enough to describe the considered class of algorithms.

Published in: *Symp. on Application Accelerators in High Performance Computing (SAAHPC)*
 DOI: 10.1109/SAAHPC.2011.12

One important novel language feature, and main difference of our work to related work, is the concept of domains and subdomains, on which operations and properties can be defined. By this powerful language construct, we enable scientists to divide a problem space into subdomains with arbitrary form, position and operations. For example, in nanophotonics an interesting nanophotonical device is given by a microdisk cavity in a perfect metallic environment. To simulate the propagation of light inside the device, we must evaluate different PDEs for different kind of materials, which are represented by our subdomain concept. Both, formulation in the DSL and mapping to different targets of such structures is achieved efficiently by our subdomain concept. During the iterative simulation process, the defined operations are applied to every element of their respective subdomain grids and are therefore referred to as *stencil operations* [3].

In the following, the transformation process is explained in detail (for an overview see Fig. 1).

In our current implementation, we define stencil operations as abstract syntax trees (ASTs) instead of using a compiler frontend (lexer and parser) to create them. They are used to create an intermediate representation in a consecutive pass. This representation includes a mapping of arbitrary 3D domains, subdomains and their corresponding properties to discretized 3D grids and subgrids. Alongside to the mentioned assignment of stencil equations and sequences of them to grids, scalars and arrays, needed by those equations are also assigned to the grids.

Then, we perform a data access pattern analysis of the stencil equations. This way, we can generically check for potential boundary access violations and optimizing memory access pattern to save bandwidth already before any code is generated or even executed. If this test is passed, we create index transformations between grids and subgrids to finalize the representation of the problem domain.

Aiming to support different kinds of target architectures and corresponding APIs and programming languages, we developed an abstract target language representation model, which is used in combination with the instance and target model to generate code for a selected architecture. In the current prototype implementation, we focus on support for NVIDIA GPU devices by generating CUDA (Compute Unified Device Architecture) code. However, in addition we already support multiple devices based on MPI in combination with CUDA with an similar approach as used in [4]. An advantage of our abstract language representation is that the gap to support targets other than CUDA is minimal.

One code generation option, investigated in this work, controls how an algorithm is transformed for efficient representation of subdomains in CUDA code. As explained, the subdomains on which stencils operate can be arbitrarily shaped. Their extension is defined by using a boolean function which can determine if a global grid point is supposed

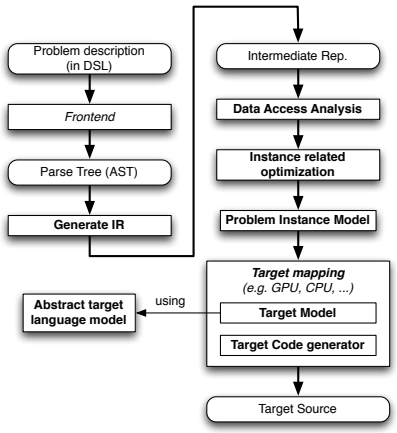

Figure 1: Implementation of our approach

to be part of the subgrid. In our prototype implementation, this algorithmic description can be transformed to actual CUDA code in three ways (see chapter IV). The numerical outcome of these codes is identical, however depending on the balance between memory throughput, caching and computational effort, the performance may be different.

III. Computational Nanophotonics (FDTD)

For demonstration of the potential of our approach and the attainable performance of the generated code, we consider in the following the Finite Difference Time Domain (FDTD) algorithm [5]. FDTD solves the Maxwell PDEs, which in turn describe the evolution of electromagnetic fields. For benchmarking the code we choose an example system consisting of a microdisk cavity in a perfect metallic environment providing specific parameters and a geometry which defines the subdomains. This is a realistic example of a nanophotonic device, which has well known analytic solutions, the so-called Whispering Gallery Modes. The cylindrical disk subdomain is modeled using vacuum FDTD update equations for magnetic- and electric fields. It is enclosed by an ideal metallic material, wherein the magnetic field is zero and only the electric field values need to be updated. Our code generator maps this kind of material compositions by sampling the physical dimensions with a user defined sampling factor to a grid and creates arrays (E_x, E_y, H_z) on the GPU and assigning scalar update coefficients (ca, cb, da, db) to the grid. The FDTD algorithms can be expressed using the following set of update equations:

$$E_x[i] = ca \cdot E_x[i] + cb \cdot (H_z[i] - H_z[i - dy]) \quad (1)$$

$$E_y[i] = ca \cdot E_y[i] + cb \cdot (H_z[i - dx] - H_z[i]) \quad (2)$$

$$H_z[i] = da \cdot H_z[i] + db \cdot (E_x[i + dy] - E_x[i] + E_y[i] - E_y[i + dx]) \quad (3)$$

Here, $i \pm dx$ and $i \pm dy$ denote neighbors of the grid cell with index i in x and y direction. Because the stencils

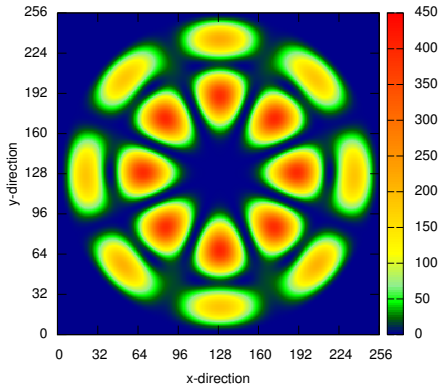

Figure 2: Time-integrated energy density for a microdisk cavity in a perfect metallic environment

operate on nearest neighbors, we define an update grid region, which guarantees, that all points can be updated, which ensures that all accessed nearest neighbors outside the update region exists. This implies that we do not need to handle border elements differently from elements inside the update region, which is an advantage for the considered target hardware architectures. To inject non-zero fields into our simulation, we extend our PDE with a point-like time-dependent inhomogeneity which physically represents an optical dipole point source. In each iteration of the simulation loop, the E and H fields are computed based on previous values in separate substeps, the point source amplitude is added at one grid point and the time-integrated energy density is computed to extract the excited mode from the simulation (an example result is given in Figure 2): $H_{zsum}[i] += H_z[i]^2$. We want to emphasize that the set of update equations can be easily extended, e.g., to model other material types like Lorentz oscillators or to perform further analysis on the computed data.

IV. Results

For evaluation of our approach, we implemented a reference CPU implementation based on OpenMP using up to 8 cores (two Intel Xeon CPUs, E5520 4-cores @2.27GHz, GCC 4.4.0, GNU Linux 2.6.18 x86_64) and compared it with the performance of code generated for the GPU system (Tesla Fermi C2050 and GTX480, CUDA 3.2) for a range of problem sizes and optimization parameters. Our MPI-CUDA code was evaluated with two nodes, connected over ethernet. As explained above we developed three different code transformation methods to represent the subdomains on which stencils operate. All choices are analyzed with respect to their performance for a range of problem sizes. Computations are done in double precision.

1) Our first approach, *No mask*, uses one or more functions to model subdomains. They are evaluated at runtime to select the appropriate stencil operation for each point in the grid and therefore imposing additional mathematical operations. Nevertheless this option allows to represent geometries in a compact form which reduces the amount of needed GPU memory and bandwidth. We achieve up to 2060 MStencils/sec by this option.
   Because threads in a CUDA-capable Fermi device are scheduled in warps (a bunch of 32 threads), it is recommended that all threads in a warp follow the same execution path to prevent a performance penalty caused by serialization. Hence, depending on the problem geometry, the *No mask* mapping may come along with a performance decrease.

For the following subdomain mapping option, we use a *Mask* to represent the affiliation of each point in the grid to a subdomain.

2) The second mapping technique is accomplished by using a *Multiplicative Mask*, which is essentially a lookup table containing the numbers 0 or 1 for each grid point. This number is multiplied by the arithmetic update-expression and therefore determines if a grid value is changed or not, hence for 1 (inside subdomian) the stencil operation is applied, while for the value 0 (outside subdomain) the value of the grid cell is not changed. Compared to the previous mapping option, we get a performance increase of up to 240 million stencil operations per second. The overhead of this technique is given by a float multiplication with the mask introduced to the stencil equation and the additional GPU memory capacity and bandwidth consumption for the mask, however no branch divergence occurs.
3) For small grid sizes (less than 512^2), our third transformation technique *Conditional Mask* performs nearly equal to the *Multiplicative Mask* method. Here, a boolean lookup table is used, and a "if" statement determines whether a stencil operation should be applied to the grid cell. For grid sizes greater than 4096^2, in comparison to *Multiplicative Mask* a constant performance increase of about 130 million stencil operations per second for double precision is achieved despite the additional branch divergence that can occur. This result is slightly unexpected and can be seen as an indicator for the complexity of the sophisticated task scheduling facilities of modern GPUs. This motivates our approach to use automated benchmarking as feedback in addition to expert wisdom to achieve the best possible optimization.

Inherently, the computation intensive part of the considered application is given by the stencil operations, which are applied to each point in the grid. Our code generator maps these parts to GPU kernels which are executed on the GPU. An application-dependent flexible utilization can

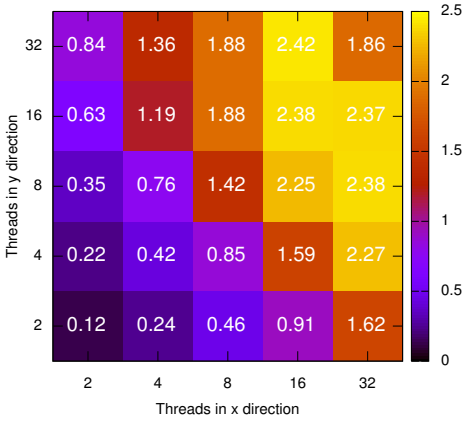

Figure 3: Giga Stencil operations per second depending on Thread Block Size for grid size 2048^2 (double precision)

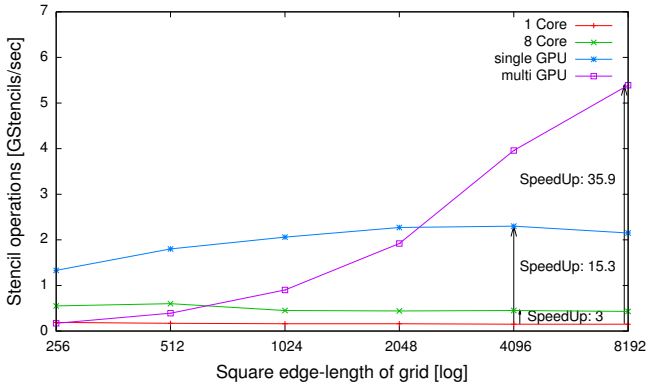

Figure 4: GStencils/sec over grid size, DP (square grids, plotted over edge length), *Multiplicative Mask* is used for GPUs

be achieved by the user defined kernel runtime-setup (parameters: threadblock- and grid-size). The reader is refered to the CUDA Programming Guide for detailed grid- and threadblock explanation.

We have analyzed a wide range of possible 2D configurations that satisfy these constraints for a grid size of 2048^2 in double precision. In contrast to a possible initial expectation that only the total number of threads is relevant, we observe a strong dependence on the thread block layout. For example, a thread block with geometry 2x32 achieves only 0.84 GStencils/sec for double precision, in contrast the 32x2 thread block layout performs almost twice as fast with 1.62 GStencils/sec. Also, note that, counter-intuitively, the highest performance is not obtained for a maximal number of threads (32x32) in a thread-block, but for the 32x16 configuration. Again, we attribute this effect on the complexity of the GPU scheduler.

The problem size determines whether it is beneficial to execute the simulation on multiple GPUs. For example, grid sizes smaller than 512^2 gain no performance benefit from the MPI-CUDA multi GPU solution due to communication costs (cutting edges of the partitioned problem must be communicated because of nearest neighbor computation). On the other hand, the code generated for a single GPU gains speedups over the whole considered grid size range. For double precision, a speedup of $15.3\times$ was achieved for the two largest grid sizes compared to a single CPU benchmark. Another important message is, that for the considered problem it is not beneficial to use a multi GPU solution for grid sizes below 2048^2 for double precision. However, for the largest considered grid sizes, the multi GPU solution accomplish speedups of $35.9\times$ compared to a single CPU.

V. Conclusion and Future Work

Based on the idea to separate the algorithm description from specifics of particular parallel computing architectures, we demonstrated the potential of this concept with a prototype implementation focused on neighbor-local memory access patterns and backends for single GPU and multi GPU targets. First results for our multi GPU code generation, an area where manual implementation is tedious and error-prone due to the complexity and several components, shows the future potential of our approach by providing a speed-up of up to $35.9\times$ compared to a single CPU without any change to the input representation. Affirmed by these results we plan to extend our framework to further hardware architectures and several optimization strategies (e.g., for complex memory accesses).

Acknowledgement

This work has been partially supported by the German Ministry for Education and Research (BMBF) under project grant 01 | H11004 (ENHANCE).

References

[1] C. Dineen, J. Förstner, A. Zakharian, J. Moloney, and S. Koch, "Electromagnetic field structure and normal mode coupling in photonic crystal nanocavities," *Opt. Express*, vol. 13, no. 13, pp. 4980–4985, Jun. 2005.

[2] V. Kindratenko and P. Trancoso, "Trends in high-performance computing," *Computing in Science Engineering*, vol. 13, no. 3, pp. 92–95, May–Jun. 2011.

[3] M. Christen, O. Schenk, and H. Burkhart, "Automatic code generation and tuning for stencil kernels on modern shared memory architectures," *Computer Science - Research and Development*, pp. 1–6, 2011, 10.1007/s00450-011-0160-6.

[4] D. A. Jacobsen, J. C. Thibault, and I. Senocak, "An MPI-CUDA implementation for massively parallel incompressible flow computations on Multi-GPU clusters," in *AIAA Aerospace Sciences Meeting and Exhibit*, vol. 48th, Orlando, FL, Jan. 2010.

[5] A. Taflove and S. C. Hagness, *Computational Electrodynamics: The Finite-Difference Time-Domain Method, Third Edition*, 3rd ed. Artech House Publishers, Jun. 2005.