Migration of SES3D application to "Piz Daint"

Executive summary

This document contains a report on the migration of a seismic wave propagation application known as SES3D to the new massively parallel heterogeneous high-performance system XC30 Cray "Piz Daint" that has been recently installed at the Swiss National Supercomputing Centre. This migration had two principal objectives: (1) improving research efficiency of computational seismologists, especially of those working in the field of seismic tomography and (2) establishing the software engineering practices and collaboration scenarios that would enable efficient utilization of computational resources of "Piz Daint" and similar emerging high-performance systems.

Our efforts resulted in building a stable SES3D implementation that, for the use cases tested, runs at least 6 times faster than the original implementation built with the best available Fortran 90 compiler. Furthermore, using GPU accelerators allowed us to achieve 3.5-4 times performance improvement over the best known homogeneous implementation on "Piz Daint". The new implementation achieves the GPU memory throughput varying from 60% to 80% of the peak bandwidth, thus providing good utilization of hardware resources. These results fully justify the CSCS strategic decision of upgrading "Piz Daint" to the heterogeneous computing architecture.

We plan to publish an article on the results of this project. Furthermore, we received an invitation to the Supercomputing Frontiers 2015 conference (Singapore).

This paper discusses various aspects of SES3D migration, in particular, choice of a programming model, software engineering techniques, impact of the heterogeneous computing model on the application architecture, utilization of GPU hardware, and collaboration scenario and role assignment of project participants.

Objectives

SES3D is a seismic wave propagation tool based on spectral elements originally developed by Prof. Andreas Fichtner [1, 2, 3]. It has been originally written in Fortran 90 and MPI and was designed for use on a wide variety of homogeneous massively parallel systems. In this project we aimed at migration of this application to the new massively parallel heterogeneous high-performance system XC30 Cray "Piz Daint" that has been recently installed at the Swiss National Supercomputing Centre. This migration had two principal objectives: (1) improving research efficiency of computational seismologists, especially of those working in the field of seismic tomography and (2) establishing the software engineering practices and collaboration scenarios that would enable efficient utilization of computational resources of "Piz Daint" and similar emerging high-performance systems.

Application

SES3D is a program package for the simulation of elastic wave propagation and waveform inversion in a spherical section of the Earth. The package is based on a spectral element discretization of the seismic wave equation combined with adjoint techniques.

The computational domain in SES3D is represented as a large regular area in 3D spherical coordinates; this area is divided into smaller disjoint blocks in each of three dimensions. The size of this block grid defines the problem size.

SES3D implements three computation modes, one mode being used for normal simulation, and two others employed for the forward and inverse adjoint calculations respectively.

Since its first release in 2007, SES3D has been running on numerous high-performance systems, including “HLRBII” and “SuperMuc” at the Leibniz Rechenzentrum in Munich, and “Monte Rosa” and “Piz Daint” at CSCS. Research published in more than 30 international peer-reviewed articles is based on SES3D simulations. Currently, SES3D is the core application of nearly 20 research projects conducted at various institutions, including the University of Michigan, LMU Munich, Istanbul Technical University, the Australian National University, GFZ Potsdam and ETH Zurich.

Hardware

"Piz Daint" is a new massively parallel heterogeneous high performance computing system of Cray XC 30 family operated by the Swiss National Supercomputing Centre (CSCS). It is featuring 5,272 computing nodes, each node being equipped with an 8-core 64-bit Intel Sandy Bridge CPU (Intel Xeon E5-2670), an NVIDIA Tesla K20x GPU accelerator with 6 GB GDDR5 memory, and 32 GB of host memory. The nodes are connected by the proprietary "Aries" interconnect from Cray.

Programming model

For this research we decided to employ a hybrid programming model based on a combination of several well-established software development technologies that include C as a host programming language, MPI 3.0 as a library for the inter-node communication, and CUDA as a toolkit for programming NVIDIA GPU accelerators. We are using compilers and libraries of Cray Programming Environments 5.1 in combination with NVIDIA CUDA Toolkit 5.5.

In our early experiments we were using the Unified Parallel C (UPC) as a communication platform. Our experience has proven however, that for the given problem domain UPC is unsatisfactory from both software performance and developer flexibility points of view.

We hold an opinion that, although sufficient for building the quality high-performance applications within reasonable time, the selected combination of mainstream programming technologies leaves a substantial room for improvement of programmer productivity. This improvement might be achieved by a single programming model that would replace the present heterogeneous mix of relatively low-level technologies thus reducing complexity of application code, improving its portability, increasing programmer productivity and being accessible for developers who are not experts in software engineering. However, to be of any practical use, such a programming model must be based on experience gained during implementation of several relevant applications using the existing programming technologies; therefore we did not focus on the programming model issues at this phase of our research.

Use cases

For this project we have chosen two SES3D use cases. Each use case is characterized by the dimensions of its computational domain. Every test run processes a number of independent events, each event representing a separate earthquake. Processing of each event requires a certain number of iterations. Time required to complete the run is nearly proportional to both these numbers.

Our first use case (code name "Turkey") has been borrowed from the original SES3D distribution and related tutorial. It is featuring a 69 x 112 x 32 grid of blocks, and a single event of 4000 iterations.

Our second use case (code name "Japan") has been inspired by the ongoing master study of Saule Zukauskaite. It is featuring a 77 x 68 x 77 grid of blocks, and 10 events of 14000 iterations each.

SES3D algorithms expose a high degree of data level parallelism and therefore the application can be conveniently implemented on massively parallel computing architectures. This implementation is facilitated by parallelization configurations that specify 3D processing grids consisting of CPU cores in homogeneous case or GPU devices (and containing computing nodes) in heterogeneous case. The whole computational domain is partitioned in all three dimensions into block groups, each group being mapped to a single element of the respective processing grid. For example, if a 69 x 112 x 32 grid of blocks is mapped to a 3 x 4 x 4 processing grid, then each element of the processing grid is assigned a group of 23 x 28 x 8 blocks.

We make distinction between "wide" and "compact" parallelization configurations. Wide configurations are characterized by a relatively large processing grid size and rather small amount of data assigned to each processing grid element. Compact configurations have a substantially smaller grid size and respectively larger size of data sets assigned to each processing element. Parallelization configurations used in our study are these:

"Turkey", homogeneous, compact: 3 x 4 x 4 (6 nodes, 48 CPU cores)
"Turkey", heterogeneous, compact: 3 x 2 x 1 (6 nodes, 6 GPU devices)

"Japan", homogeneous, compact: 11 x 1 x 7 (10 nodes, 77 CPU cores)
"Japan", heterogeneous, compact: 11 x 1 x 1 (11 nodes, 11 GPU devices)

"Japan", homogeneous, wide: 11 x 17 x 7 (164 nodes, 1309 CPU cores)
"Japan", heterogeneous, wide: 11 x 1 x 11 (121 nodes, 121 CPU cores)

We expected originally that wide configurations favor homogeneous (CPU-based) versions since smaller data sets fit CPU caches better. On the other hand, it was expected that compact configurations provide more benefits for heterogeneous (GPU-based) configurations because GPU devices achieve better performance processing reasonable large data sets that suit better for amortization of latencies in the GPU streaming processors. As we will demonstrate later in this document, performance figures fully agree with our original expectations.

Software engineering aspects

We have based the migration process on an assumption that transition to a heterogeneous computing model will have impact on all aspects of the application architecture. This means, in particular, that it would not be sufficient to just extract the most critical parts of code and re-implement them as optimized CUDA kernels. Instead, the entire application architecture has to be carefully re-engineered in order to utilize the potential of heterogeneous hardware efficiently. The main reason is that any inefficiency in the serial part of the application can easily kill any benefits of using GPU.

SES3D is managing large volumes of data arranged as a number (about 100) of large arrays. Interchange of these data volumes between GPU memory and CPU memory of respective computing nodes at every iteration would incur large performance overhead. Due to the nature of the algorithm, this interchange cannot be done asynchronously. Therefore, in order to use GPU capabilities efficiently, it is necessary to host all data arrays on GPU permanently and therefore implement all computations on GPU. The role of corresponding host CPU should be limited to input and output operations, inter-node communication, and management of processing workflow. We made the only exception to the initialization procedure that takes place once per event and has performance overhead that is well amortized by the big number of computation iterations. We thus try to keep data interchange between GPU and surrounding environment at the absolutely necessary minimum. (It is worth noting that with this approach we use only one CPU core out of eight available on every computing node of "Piz Daint". It is a subject for the special discussion whether it would make sense to find employment for the remaining seven cores; this discussion is however out of scope of this document.)

Data interchange between the computing nodes represents an essential part of the computation algorithm. In terms of performance it causes certain overhead which might offset benefits provided by GPU. It is essential therefore to tune inter-node communication carefully to keep data interchange between GPU devices of different nodes reasonably fast.

Computation algorithm for the adjoint calculations requires regular transfer of large data volumes between the application and external storage. These transfer operations are therefore highly disruptive to our strategy of minimizing data interchange between GPU and other hardware components. Therefore the special methods had to be designed in order to reduce the overhead of external read and write operations and also perform data interchange between GPU and host CPU memory asynchronously in parallel with main computations.

It has been proven that choice of parallelization configuration plays a critical role. In particular, compact configurations are the most beneficial for heterogeneous systems. However, absolute time required for processing of a single event is still significantly higher than time required by the wide configuration. This observation led to introduction of parallel processing of events in order to utilize in practice performance advantages of compact heterogeneous configurations.

Initial code rewriting

We started migration with porting the original Fortran 90 code to C. During this porting we made various code transformations. In particular, number of dimensions of all principal arrays has been reduced from 6 to 2; vectorized operations of Fortran 90 transformed into explicit loops; substantial amount of redundant computations within the loops discovered and optimized. These transformations alone significantly improved application performance compared to the original Fortran 90 version.

Introduction of explicit loops helped to analyze data parallelism essential for mapping computations to GPU. Based on this analysis the initial set of GPU data layouts and CUDA computational kernels has been designed and implemented.

Optimization of inter-node communication

The SES3D algorithm requires inter-node data interchange within every iteration. The interchange pattern is regular and conceptually quite simple, however its correct and efficient implementation has proven to be non-trivial. The original version provided a simplified implementation that reserved large temporary buffers in memory and therefore required copying of substantial data volumes at each iteration. We completely re-engineered the data interchange part and created a carefully tuned (albeit somewhat complex) implementation.

Like the original code, the new implementation relies on MPI as a principal communication platform. However, after the due performance study, we decided to replace the classic MPI point-to-point communication model used in the original code with one-sided communications based on the MPI Remote Memory Access (RMA) communication mechanisms, which provide better performance for all parallelization configurations.

Optimization of CUDA kernels

We employed the well-known programming techniques for optimization of the application CUDA kernels such as tuning the layout of data arrays in order to enable coalesced access to device memory or using GPU shared memory to reduce number of redundant device memory accesses. Whenever feasible, we implement highly optimized re-usable kernels for commonly used functions like array reductions. We extensively used the NVIDIA profiler (*nvprof*) to measure performance of individual CUDA kernels as well as data transfers between CPU hosts and GPU devices.

We deliberately decided to avoid any heroic efforts aimed to squeeze the last bits of performance out of GPU. We justify this decision by our assessment of performance limits of CUDA kernels. SES3D is a typical memory-bound application, therefore performance of its CUDA kernels is constrained by the peak memory bandwidth of the GPU device. For NVIDIA K20x this bandwidth equals 250 GB/s. Profiling of our kernels reports the GPU device memory throughput varying between 160 and 200 GB/s, that is, between 60% and 80% of the peak bandwidth. Throughput of 80% of the peak is usually considered very good, therefore it is unlikely that the best kernels can be significantly improved. Improvement of less performing kernels in order to increase their throughput from 60% to 80% might be feasible, however the net effect is expected to be small and not worth an effort provided that slower kernels represent only a part of all CUDA kernels and GPU computations represent only a part of all application activities.

Performance results

We use simple yet practical methodology for performance assessment. The wall clock time for each configuration and test run is measured. Based on this figure, the number of iterations processed per minute per computing node is calculated.

Performance for "Turkey" use case has been measured for both CPU and GPU versions in the compact parallelization configuration and the normal simulation mode only. At each test run a single event with 4000 iterations was processed. The following average wall clock times have been recorded:

CPU, 3 x 4 x 4 (6 nodes):	7 min
GPU, 3 x 2 x 1 (6 nodes):	2 min

Thus the speedup of the heterogeneous (GPU) over homogeneous (CPU) version achieved for this case is approximately 3.5 times.

Performance of "Japan" use case has been measured for both CPU and GPU versions as well in both compact and wide parallelization configurations. At each test run 10 events were processed in a sequence with 14000 iterations for each event. Separate tests were conducted for each of three supported computation modes. In the text below these modes are designated via an *adjoint_flag* parameter, which is assigned 0 for normal simulation, 1 for adjoint forward, and 2 for adjoint inverse computations respectively. The following wall clock times have been recorded in one of the representative test series (results expressed as hours:min:sec).

CPU, compact, 11 x 1 x 7 (10 nodes)

adjoint_flag = 0	7:00:16
adjoint_flag = 1	7:12:10
adjoint_flag = 2	7:18:39

GPU, compact, 11 x 1 x 1 (11 nodes)

adjoint_flag = 0	1:18:24
adjoint_flag = 1	1:49:22
adjoint_flag = 2	1:52:49

GPU, wide, 11 x 1 x 11 (121 nodes)

adjoint_flag = 0	0:16:24
adjoint_flag = 1	0:20:42
adjoint_flag = 2	0:31:10

CPU, wide, 11 x 17 x 7 (146 nodes)

adjoint_flag = 0	0:21:32
adjoint_flag = 1	0:22:40
adjoint_flag = 2	0:26:46

The following performance figures (iterations / node / minute) have been calculated for the normal simulation mode (adjoint_flag = 0):

GPU, 11 x 1 x 1:	140000 / 11 / 78.5 = 162.1
GPU, 11 x 1 x 11:	140000 / 121 / 16.5 = 70.1

CPU, 11 x 1 x 7:	140000 / 10 / 420.5 = 33.3
CPU, 11 x 17 x 7:	140000 / 164 / 21.5 = 39.7

These figures fully support our original assumption stating that wide parallelization configuration is more beneficial for homogeneous (CPU) versions while compact configurations yield better performance for heterogeneous (GPU) versions.

Furthermore, by comparison of the best performances for CPU and GPU cases, the following figure for the speedup achieved on the heterogeneous architecture can be obtained:

Best GPU / best CPU = 162.1 / 39.7 = 4.1 times

Performance of adjoint calculations

The computation algorithm for the adjoint calculations, both forward and inverse, requires substantial amount of data interchange between the application and external storage. This interchange takes place once per a pre-defined number of iterations (for example, once for every 20 iterations in our "Japan" use case). In the forward adjoint mode data are written to rather large external files; in the inverse mode they are read from these files in the reverse order. Each interchange requires reading or writing of 9 large arrays; the array size is proportional to the amount of application data assigned to a single processing element. The volumes to be interchanged thus highly depend on the parallelization configuration. For example, the size for a single array read or written in "Japan" use case is 154 KB for the wide homogeneous and 18,326 KB for the compact heterogeneous configurations respectively. (These data volumes are inversely proportional to the respective number of processing elements of 1309 and 11 used in the respective configurations.)

These interchange operations add certain penalty to the overall performance. This penalty is insignificant in wide homogeneous configurations but increases dramatically in compact configurations, especially in heterogeneous cases. There are two distinct sources of this increase: (1) increased volume of data to be interchanged between the application and external storage in the compact configuration and (2) increased volume of data to be interchanged between host and device memory in the heterogeneous configuration. These two sources have to be analyzed and dealt with separately.

Our performance study of interchange with the external storage suggests that the related penalty is tolerable if data are read or written sequentially. Unfortunately, this is not a case in the inverse adjoint mode when data are read in the reverse order. The original un-optimized application version demonstrated disappointing wall clock time of more than 3 hours for the heterogeneous compact configuration in "Japan" use case. To solve this problem we introduced buffering of read operations in the inverse adjoint mode. For each array to be read the application allocates a buffer large enough to hold about 30 consequtive array instances which are pre-fetched via a single read request. This approach allowed us to reduce the run time and make it comparable to that of forward adjoint mode.

However, as performance figures listed above suggest, there is still a large (40%) run time difference between the normal simulation and adjoint calculations for the compact heterogeneous configuration. Profiling shows that the major portion of this difference is related to the data interchange between host and GPU device memory. This interchange can be done in parallel with the main computations and thus the related overhead can be amortized by implementing a separate CUDA stream responsible solely for asynchronous moving of these data between host and device memory. Implementation of this feature in progress now.

Parallel processing of events

The initial implementation of SES3D processed all events sequentially, that is, at each moment all computing nodes were involved in processing of the same event. This sequential scenario appears impractical for the compact parallelization configurations. These configurations provide the superior performance in heterogeneous cases, however, the absolute wall clock time required to process all events sequentially is too large. On the other hand, the number of computing resources involved in computations for the compact parallelization configurations is substantially (usually in order of magnitude of more) smaller than in case of wide configurations. Therefore, if comparable number of computing resources is available for both compact and wide configurations, then the compact configuration may utilize these resources in full by splitting them into smaller groups, then distributing all events to be processed evenly among these groups and allowing them to process events in parallel.

For example, performance figures listed above suggest that a wide homogeneous configuration of 164 computing nodes requires about 2.15 minutes to process a single event. For a compact heterogeneous configuration of 11 nodes the respective time constitutes 7.85 minutes. Let us assume that there are 60 events to process. The wide homogeneous version will require approximately 2.15 x 60 = 129 minutes to complete the work. In the heterogeneous compact version it is possible to create 15 groups of 11 computing nodes each (thus using 165 nodes in total) for parallel processing of events. Each group becomes responsible for processing of just 4 events, therefore the total processing time in this case is 7.85 x 4 = 31.4 minutes. This approach allows to achieve in practice the aforementioned 4-times speedup over the wide homogeneous configuration using comparable (165 vs. 164 computing nodes) amount of hardware resources.

We implemented the respective upgrade of SES3D architecture by adding support for parallel event processing. If a number of processing elements allocated at application start is a multiple of a number specified by a parallelization configuration, the application automatically splits available processing elements in groups and partitions a list of events into chunks, assigning each chunk to its own group for parallel processing.

Code portability

We are using the same code base for both homogeneous and heterogeneous versions of the application. Condition compilation directives of C pre-processor are used to choose the version at application build time. This approach in combination with our reliance on the industrial-standard programming technologies makes the code widely portable among various high-performance architectures, both homogeneous and heterogeneous equipped with NVIDIA GPU of Kepler family.

It is worth noting that substantial code duplication takes place since the same computation algorithms have been implemented twice, one time as serial C code and another time as code of GPU CUDA kernels. This code duplication is regarded as a significant disadvantage, which cannot be efficiently addressed within the chosen programming model. A possible solution could be obtained by introduction of a programming model with the higher degree of abstraction that would allow automated generation of serial as well as CUDA code from a single common high-level algorithm description. We plan to address implementation of such a programming model in our future research.

Collaboration scenario

Traditionally in the academic environment a problem domain specialist (or a small group of such specialists) had been responsible for the entire life cycle of a research application including design of numeric algorithms, coding, debugging, testing, and maintenance. This scenario is becoming less feasible with an advent of innovative heterogeneous high-performance computing systems because of the disruptive nature of changes these systems cause in hardware architecture and programming models. Computers of this class are exceptionally powerful but also very expensive and substantial software engineering skills are required to use their resources efficiently. To address this issue introduction of a new, more industrial-like collaboration scenario based on a more clear separation of roles played by project participants is required.

In this project we have introduced three distinct principal roles: numeric analyst, software engineer, and power user. A numeric analyst is responsible for the formulation of numeric algorithms and implementation of a working application prototype in whatever programming language suitable for this purpose. A software engineer transforms this prototype implementation into production quality code applying whatever means necessary to improve the code structure and performance on a given target architecture. It is expected that in course of this transformation certain good engineering practices applicable for a given problem domain will be established. A power user recurrently applies the evolving implementation to a selected set of use cases and provides the software engineer with feedback regarding correctness, performance, and usability of the application. We expect also that at later stages the power user will become an "application guru" being in charge of assisting the new users and collecting user feedback needed for the further application improvement.

We have successfully implemented this collaboration scenario in the given project. The roles of numeric analyst, software engineer, and power user had been assigned to Prof. Andreas Fichtner, Alexey Gokhberg, and Saule Zukauskaite respectively.

Further steps

We plan to re-use the experience, engineering practice, and collaboration scenarios developed in course of this project for implementation of other seismology applications of interest. The most likely next project will be focused on porting to "Piz Daint" of the Parallel Noise Correlation Toolbox (PANOCO) that is currently under development at our group.

We plan to publish an article on the results of this project. Furthermore, we received an invitation to the Supercomputing Frontiers 2015 conference (Singapore).

References

[1] Andreas Fichtner, Heiner Igel, 2008.
Efficient numerical surface wave propagation through the optimization of discrete crustal models - a technique based on non-linear dispersion curve matching (DCM)
Geophysical Journal International, 173, 519-533.

[2**]** Andreas Fichtner, Brian L. N. Kennett, Heiner Igel, Hans-Peter Bunge, 2009.
Full waveform tomography for upper-mantle structure in the Australasian region using adjoint methods
Geophysical Journal International, 179, 1703-1725.

[3] Andreas Fichtner, Jeannot Trampert, Paul Cupillard, Erdinc Saygin, Tuncay Taymaz, Yann Capdeville, Antonio Villasenor, 2013.
Multi-scale full waveform inversion
Geophysical Journal International, 194, 534-556.