Directive-based Heterogeneous Programming
A GPU-Accelerated RTM Use Case

Stéphane BIHAN
Georges-Emmanuel MOULARD
Romain DOLBEAU
CAPS entreprise
4 allée Marie Berhaut
Rennes, 35000, France
www.caps-entreprise.com

and

Henri CALANDRA
Rached ABDELKHALEK
TOTAL Technical and Scientific Center
Avenue Larribau
Pau, 64000, France
www.total.com

Monday 25th May, 2009

Abstract

Hybrid parallel multicore architectures based on Graphics Processing Units (GPUs) can provide tremendous computing power. Current NVIDIA and AMD Graphics Product Group hardware display a peak performance of hundreds of gigaflops. However, exploiting GPUs from existing applications is a difficult task that requires non-portable rewriting of the code. In this paper, we present HMPP, a Heterogenous Multicore Parallel Programming workbench with compilers used to accelerate a Reverse Time Migration (RTM) application in a unintrusive manner while preserving the legacy code.

1 Introduction

Using Graphics Processing Units (GPUs) for scientific computing is a recent and fast evolving trend [1]. The evolution has been so fast that the usual general-purpose computing on graphics processing units (GPGPU) that usually refers to programming vertex and fragment shaders, is now obsolete.

Many phenomena have been at the origin of the use of GPUs in scientific computing. The first one is the evolution toward multicore architecture that doubles the number of cores instead of doubling clock frequency every 18 months. As a consequence, this has driven a programming effort toward parallel programming and has offered opportunities to hardware accelerator based approaches. Performance of GPUs are about 750 times higher than a decade ago.

The second one has been the introduction of programmable vertex and fragment shaders [4] that have exposed a very high potential computing power to programmers outside the graphics area.

The third phenomenon is new programming languages like NVIDIA CUDA, Brook+ or soon OpenCL that are based on a stream model more suited to scientific programming than the obscure OpenGL or DirectX standards used by GPGPU pioneers.

However, the main objective of these languages is to expose some of the specifics of the stream architecture in order to better exploit their performance. While CUDA and Brook+ are vendor-specific programming languages, the new OpenCL initiative aims at becoming a programming standard for a large range of available accelerators.

Based on compiler directives, HMPP (Heterogeneous Multicore Parallel Programming) offers a higher level of abstraction but still allows developers to fine tune the programming of accelerators. HMPP provides developers with a heterogeneous C and Fortran compiler with

hardware-specific code generators that translate C and Fortran functions in CUDA or SSE . Hardware-specific codings are dissociated from the legacy code as additional software plugins. Contrary to applications that have been specifically written for a target architecture, HMPP produces applications that run on various hardware platforms whether an accelerator is present or not.

The extraordinary challenge that the oil industry must face for hydrocarbon exploration requires the development of leading edge technologies to recover an accurate representation of the sub surface. Over the last 20 years the industry has seen a fantastic leap in the ability to process seismic data and to build an increasingly accurate image of the Earths structure. Seismic Depth imaging and High Performance Computing are both key components of this evolution.

Indeed the fast evolution of computers has enabled the development of specialized algorithms allowing the processing of increasingly large volumes of data generated by seismic acquisitions. Among these technologies, first arrival Kirchhoff, like integral methods has difficulties in imaging complex geological structures where multi pathing occurs. Downward-continuation algorithms, based on one-way wave equation factorization can only propagate partial information and cannot image very complex geological structures.

RTM based on the full wave equation discretization overcomes those limitations. Unfortunately, RTM is highly compute intensive, more than the well-established one-way wave equation methods. As such RTM is a really good application candidate for GPU acceleration.

The paper is organized as follows: section one describes the HMPP Workbench used to program and build the accelerated version of the RTM application. Section two emphases on the porting and optimization of the RTM in order to fully exploit the GPU capabilities. Section 3 reports performance results that we compare against the native version of the RTM application.

2 HMPP Workbench

HMPP proposes a solution to not only simplify the use of hardware accelerators (HWA) in conventional general purpose applications but also to keep the application code portable.

The goal is to integrate the use of HWAs rather than porting the application to make use of them. The chosen programming approach is similar to the widely available OpenMP standard but designed to handle HWAs. The hardware-specific versions of the computations to be offloaded to a HWA are dissociated from the native application source code. As such, HMPP makes a programming *glue* between hardware-specific codings and standard programming languages.

Based on a set of directives, the HMPP Workbench provides developers with C and Fortran compilers and a runtime. It gives programmers a simple, flexible and portable interface for developing parallel applications whose critical computations are distributed, at runtime, over the available heterogeneous cores.

2.1 Directives-based Programming

2.1.1 Codelet Concept

HMPP is based on the concept of codelets, functions that can be remotely executed on a HWA.

A codelet has the following properties:

1. It is a pure function.
2. Its return value is `void`.
3. It does not contain `static` or `volatile` variable declaration.
4. The parameters are not `vararg`.
5. It is not recursive.
6. Its parameters are not aliased and can be copy-in or copy-out.
7. It does not contain `callsite` directives (i.e. remote procedure call to another codelet).
8. It does not contain any function calls such as library functions like `malloc`, `printf`, ...

Except for the aliasing property, all of these restrictions are checked by the HMPP compiler.

2.1.2 Directives for Declaring and Executing a Codelet

The HMPP directives address the remote execution (RPC) of a codelet as well as the data transfers to and from the HWA memory if different from the host CPU memory.

By default, all the parameters are loaded in the HWA just before the RPC, and the main memory is updated when the RPC has completed. More directives are provided to upload and download data to and from HWAs before the remote execution of a codelet.

All the directives belonging to the declaration, execution, data transfers, etc. of a codelet are identified by a unique label. Below is the most trivial way of accelerating an application using only two directives: a `codelet` directive to declare a function as a codelet, and a `callsite` directive inserted before the function call to specify the potential use of the codelet.

In the example Figure 1, the `matvec` function is declared as a candidate for CUDA hardware acceleration. HMPP generates the CUDA code from the C function.

```
#pragma hmpp simple codelet, args[outv].io=inout, target=CUDA
static void matvec(int sn, int sm,
                   float inv[sm], float inm[sn][sm],
                   float *outv)
{
  int i, j;
  for (i = 0 ; i < sm ; i++) {
    float temp = outv[i];
    for (j = 0 ; j < sn ; j++) {
      temp += inv[j] * inm[i][ j];
    }
    outv[i] = temp;
}

int main(int argc, char **argv) {
    int n;
  ........
#pragma hmpp simple callsite, args[outv].size={n}
     matvec(n, m, myinc, inm, myoutv);
  ........
}
```

Figure 1: `codelet` and `callsite` HMPP Directive Use Example.

The `args` parameter of the directive indicates that the `outv` parameter is used as input and output. By default, all parameters are input.

In this example, the device allocation, data upload, codelet execution and result download are performed at the call site. If the codelet is called in a loop, this leads to overhead that might inhibit the performance offered by HWAs.

2.1.3 Data Transfers Directives to Optimize Communication Overhead

When using a HWA, the main bottleneck is often the data transfers between the HWA and the main processor. To limit the communication overhead, data transfers can be overlapped with successive executions of the same codelet by using the asynchronous property of the HWA. For this, three directives can be used:

1. The `allocate` directive locks the HWA and allocates the needed amount of memory.

2. The `advancedload` directive prefetches data before the remote execution of the codelet. Moreover, if the data variable is declared constant (const qualifier in the directive parameter) then it is loaded only once for all codelet executions and reused as long as the HWA is not released.

3. The `delegatedstore` directive desactivates the default download of results from the `callsite` directive and explicitly transfers them where the directive is inserted. This allows to transfer data only once after multiple execution of a codelet, in a loop for instance.

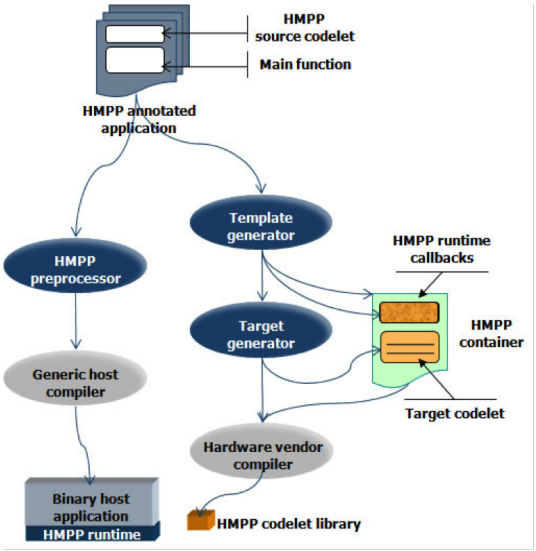

Figure 2: HMPP Compilation Flow.

It is therefore possible to perform device initialization, memory allocation and upload of input data only once outside a loop and not each iteration. Results can also be downloaded outside the loop using the `delegatedstore` directive.

2.2 Compiling Hybrid Applications

In terms of use, the HMPP compiler workflow is really close to traditional compilers with two main passes as illustrated in Figure 2: one that builds a standalone application running on the host processor and a second pass for producing the accelerated versions of the codelets as dynamic shared libraries.

2.2.1 Host Application Compilation

As shown in the left part of Figure 2, the HMPP preprocessor translates the directives into calls to the HMPP runtime in charge of managing the execution of the codelets. The preprocessed application is then compiled using the generic host compiler. Note that the host application can run standalone without hardware accelerators.

2.2.2 Codelet Generation

The right part in Figure 2 illustrates the production of the codelets as shared libraries in order to be loaded by the HMPP runtime. The template generator produces the codelet skeleton that contains all the runtime callback functions such as HWA allocation, data transfers, etc. The developer can either write the codelet kernel in the hardware programming language or let the target generator translate the C or Fortran function in CUDA or SSE. The codelet binary is finally produced using the hardware-vendor compiler. More over, the codelet generation can be further improved with the use of codelet generation directives.

2.3 Hybrid Application Execution

Hardware resources management is a critical issue that is not tackled by current exploitation systems. Supercomputers have been avoiding this problem by running only one application at a time or by partitioning the machine nodes. Unfortunately, programming and resource allocation cannot be considered separately. Sharing a GPU between applications, when it does work, usually results in very poor performance due to context switching on the device.

Current available GPUs leave to the user the sharing of a given device. This is fine when the nodes of an HPC machine are partitioned among users running a single application. However, in less controlled environments (HPC workstations, laptops) this is an issue that has to be addressed at run-time level.

The HMPP runtime handles the execution of codelets for different and various hardware (GPUs, SIMD units, FPGAs) but also for specific execution contexts. The appropriate execution hardware is selected at runtime depending on the system configuration, the resource availability and data dependent conditions. If a hardware accelerator is not present or not available, HMPP runs the native codelet function on the host system instead. If a codelet is attempting to run on a hardware, let say GPU, that is locked by another codelet, HMPP will execute it in the second GPU if there is one, then in the second target the codelet has been declared for, if any, or lastly in the host system otherwise.

3 Reverse Time Migration Acceleration

This section first gives an overview of the RTM algorithm and then describes the porting of the accelerated version using HMPP and the optimization that have been applied to exploit the computation capabilities of GPUs.

3.1 Overview of the Reverse Time Migration Computations

In RTM the two-way wave equation is solved for both the source and receiver wave fields. This is followed by an imaging condition, essentially a point-by-point multiplication of the receiver and source wave fields (i.e., zero-lag cross-correlation) which are summed over time and over shot positions.

As described in Figure3, for every shot position the general procedure consists in 3 different steps:

1. Forward sweep.
2. Backward sweep.
3. Imaging condition.

Steps 1 and 2 are based on the resolution of the full wave equation. A shot is composed by the seismic source and receivers, the forward sweep solves the full wave equation from time $t0$ up to $tmax$ with the seismic source as second member term. The backward sweep solves the full wave equation from time $tmax$ down to $t0$ with the receiver wave-field as second member. The imaging condition is then applied to the forward backward wave field at the same time t.

Because forward and backward sweep are solved in opposite time direction we need to store the wave-field during the forward sweep step. This stored wave-field is read back during the backward sweep step in order to provide the forward and backward wave-field at the same time t to the imaging condition.

By far the most computationally intensive component of RTM is the code used for solving the wave equation. We use an explicit finite difference procedure for the 3D equation in a heterogeneous medium. The procedure is 2nd order accurate in time and 8th order accurate in the spatial variables. Appropriate initial conditions (for shot and receiver waves) are given, and absorbing boundaries (or a free-boundary for the top surface) are specified.

The RTM code can be parallelized at 3 levels:

1. Shot-receiver pairs handled in a data-parallel fashion;
2. Overlapped domain decomposition in the physical domain through MPI (Figure 4);

```
for each shot (s,({ri},i=1,n))
      1/ solve forward sweep and store wave-field
          for each time step t=t0 ... tmax-1
                  Compute wavefield at time step t+1 with s as second member
                  Store wavefield at time step t
          end foreach
       2/ solve backward sweep and apply imaging condition
           for each time step t= tmax ... t1
                        Read wavefield at time step t
                        Compute image condition at time t
                  Compute wavefield at time step t-1 with ({ri},i=1,n)  as second member
          end for each
       3/ update global image
end for
```

Figure 3: RTM Computation Steps.

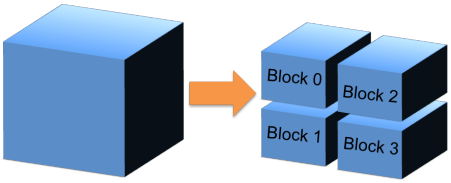

Figure 4: Subdomain Decomposition. Each subdomain is assigned to one CPU, ghosts nodes are exchanged using MPI message passing libraries.

3. Vector-parallel calculations of Laplacian, and time-updates using a shared-memory paradigm.

The 3D version of RTM uses an overlapped domain decomposition procedure, which distributes the simulation grid for each shot and receiver across multiple processors. Because each processor can operate on just a portion of the total simulation grid, the amount of memory, disk space, and computation per processor can be greatly reduced. Processors exchange overlapping zone or ghost nodes using the Message Passing Interface (MPI). Asynchronous communication MPI send and receive are used to load balance the computations and the communications between sub-domains.

3.2 CUDA Acceleration with HMPP

The RTM Fortran application was already parallelized with MPI in order to decompose one full data domain in subdomains where the computations are distributed over the different cores and nodes of a cluster machine. The subdomain decomposition is indicated at application launch to scale the RTM execution to the machine configuration.

As shown in Figure 5, each subdomain defines a border of constant width where a mirroring condition is applied in the case of the blue border and where data are sent from one subdomain to its adjacent subdomain in the yellow border.

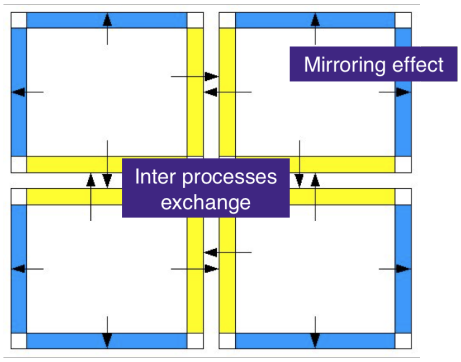

Figure 5: RTM Domain Decomposition.

One main function that does all the computations of one subdomain has been defined as a CUDA codelet with the HMPP directives. The CUDA generator of HMPP was not available at that time and the CUDA codelets were directly written. Depending on its location in the domain, the codelet performs the required subdomain computations in an NVIDIA GPU. The codelet contains the forward and backward operations of the wave field propagation. It implements two different variants: one that records the overall wave field in the CPU each time stamp of the forward sweep and another one that only records the wave field boundaries. In that last case, the codelet re-computes the wave field during the backward sweep while the wave field is sent back from the CPU to the GPU in the record wave field variant.

Two different codelets apply to the 2D and 3D versions of the RTM application. In the 3D case, the number of data to process drastically increases. The host to GPU transfers are even more expensive, inhibiting the acceleration gain we get in the 2D version. A 100x100x100 3D subdomain exchanges a lot more points than a 100x100 2D domain. This implies that for the same data point of subdomain, more memory, non-contiguous, accesses are performed.

3.2.1 CUDA Kernels Optimizations

The objective of the CUDA kernel optimizations consisted in reducing as much as possible the cost of the GPU memory accesses. Different implementations have been experimented. The first one makes a better use of the different GPU memory types like textures which are cached memory. The second one uses a simple 3D cache blocking in order to optimize data reuse.

But finally, the 3D technique has showed less efficiency than a 2D cache blocking technique sliding on the 3rd dimension. This approach allows for a larger slice and consequently a better re-use of the data. Table 1 reports the number of read accesses from GPU memory per data point for each used technique.

RTM 2D	4
RTM 3D with texture	29
RTM 3D cache blocking	7.5
RTM 3D with sliding 2D cache	4.1

Table 1: Number of Read Accesses per Data Point

3.2.2 Data Transfers Overlapping

While the CUDA kernel has been optimized to reduce the number of memory accesses per data point computation, still data transfers between the host and the GPU spending about 75% of the execution time need to be optimized.

The strategies were fourth fold going from algorithm changes to programming tricks and hardware improvement. The latest NVIDIA Tesla server upgrades PCIe to 2.0. The memory bandwidth doubles from about 3 GB/s to 6 GB/s. The programming trick consisted in exploiting the DMA-accessible pinned memory to improve host to GPU data transfer speed.

Data transfers reduction has also been achieved by offloading the mirroring condition computation in GPU, adding more inexpensive computation to the GPU side while removing partial data transfers. The last algorithm optimization consisted in overlapping computations in the CUDA kernel with data transfers. The asynchronous properties of the latest Tesla hardware has permitted such optimization.

As shown in Figure 6, this technique consists in dividing each computation and transfer iterations in a number of streams. Once the first computation stream has executed, the second one is executed in parallel with the first data transfer stream that preload data for the third computation stream, and so on.

This technique has permitted to hide a large amount of the data transfers. Execution time spent in tranferring data fell from 32% to 15% in the forward pass using one GPU and from 41% to 25% in the backward pass.

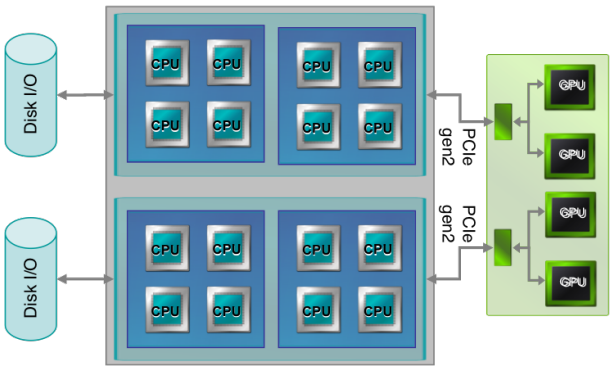

Figure 7: Accelerated Cluster Configuration.

4 Experiment Results

In this section, we report the performance achieved for different domain sizes using one or two GPUs.

4.1 Machine Configuration

Figure 7 represents a two-nodes machine connected to an NVIDIA Tesla S1070 server with four GPUs. The host nodes are bi-socket quadcore with 2x16 GB memory running at 2.5GHz. The Tesla server is connected using two PCIe 2.0 links and each GPU has 4 GB memory. Performance peak is 4 TFlops Single Precision per server and about 400 GFlops Double Precision.

4.2 Performance Results

Figure 8 gives the execution time of 3D versions of RTM running different domain sizes. We compare four different executions: two natives that uses one or two nodes, i.e four or five CPUs, in parallel using MPI and OpenMP, with GPU-accelerated versions using one or two GPUs. The more number of elements the RTM application processes, higher is the performance gain between the native version and the accelerated one.

We can notice that the accelerated version scales less efficiently using one or two GPUs than the native version running 4 to 8 CPUs. This is due to the extra cost of the data transfers between the two GPUs that need to go through the host server. Finally, if we compare the GPU-accelerated version with the 8-cores, we achieve a 3.3 speedup.

The Figure 9 shows the scaling performance of the RTM using up to 16 cores with GPU acceleration or not. If we compare the sequential version running one CPU (i.e one subdomain) against the same version but GPU-accelerated, we get a 10x performance improvement.

Our latest experiments with a 3D modeling application show that a machine with 8 NVIDIA Tesla servers S1070 (32 GPUs) is equivalent to 4.4 CPU-only machines built with 512 Intel Hapertown cores at 3.0 GHz.

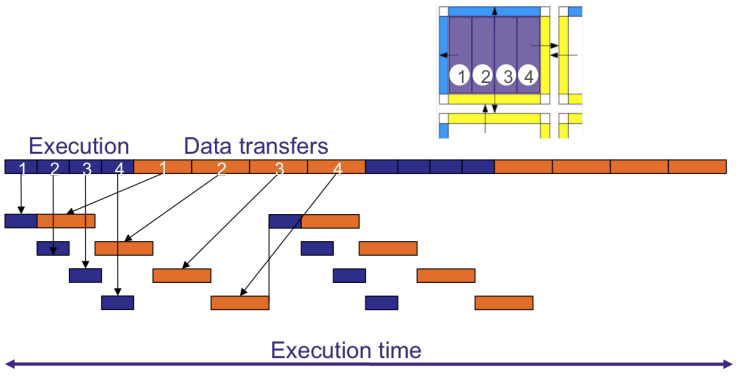

Figure 6: CPU-GPU Kernel Execution and Data Transfers Overlapping.

This result illustrates how GPU-based servers can save floor space and enery for large supercomputers.

5 Conclusion

GPU computing coupled with general-purpose multicores are very cost-effective and promising high performance heterogeneous hardware platforms. However they represent today the perfect example of the dilemma developers have to face. GPUs can provide a 10x and more performance improvement but their programming is still quite challenging.

HMPP addresses code portability as well as performance on heterogeneous multicore platforms. As a standard directive-based programming workbench, it ensures code portability, offers a high level of abstraction for programming the GPUs and is open to fine tuning GPU programming by either plugin written kernels or library functions.

Main benefits of using HMPP in the RTM application is the direct integration of CUDA hardware-specific programming in Fortran without any C or C++ glue. HMPP offers an insulation between fast evolving CUDA code and robust Fortran algorithm implementations.

We showed quite promising performance gain using GPUs to accelerate the RTM. However, this has been at the price of implementing CUDA kernels that make an efficient, and quite complexe, use of the memory and also at the price of searching for innovative techniques in order to hide the data transfers and kernel execution. For scientific applications, this requires the developer to get a too much detailed knowledge of the hardware architecture.

HMPP Workench has been enhanced with generators that automatically translate C or Fortran codelet functions in CUDA, thus hiding as much as possible hardware details while still offering an open platform. Future work can also be done in implementing real hybrid computations that make use of both CPU and GPU at the same time.

References

[1] General-purpose computation using graphics hardware, http://www.gpgpu.org/.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. AFIPS, vol. 30*, 1967.

[3] Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha. Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 3, Washington, DC, USA, 2005. IEEE Computer Society.

[4] E. Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 55–55, New York, NY, USA, 2001. ACM.

[5] By Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics, Volume 38, Number 8, April 19*, 1965.

[6] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[7] Vasily Volkov and James Demmel. Lu, qr and cholesky factorizations using vector capabilities of gpus. Technical report, Electrical Engineering and Computer Sciences University of California at Berkeley Technical Report No. UCB/EECS-2008-49, 2008.

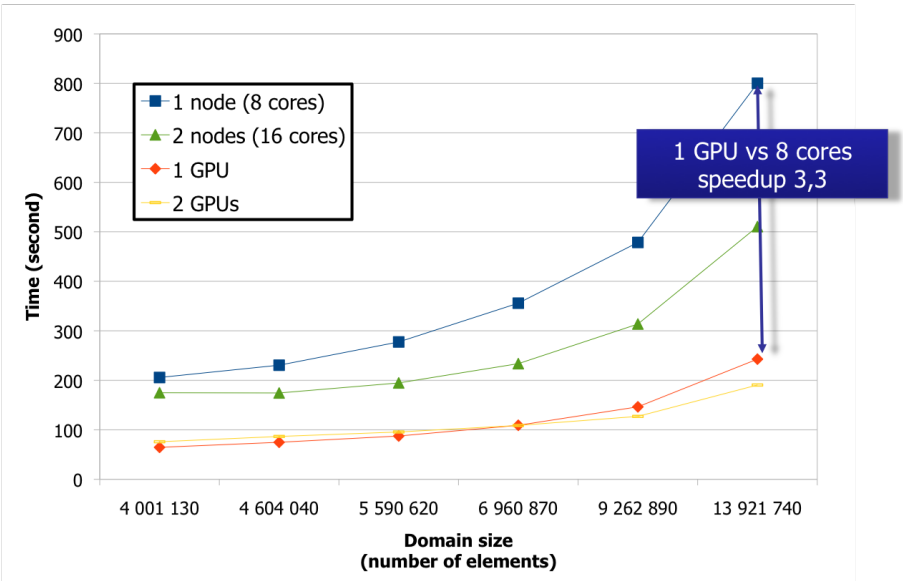

Figure 8: CPU-GPU Global Optimization.

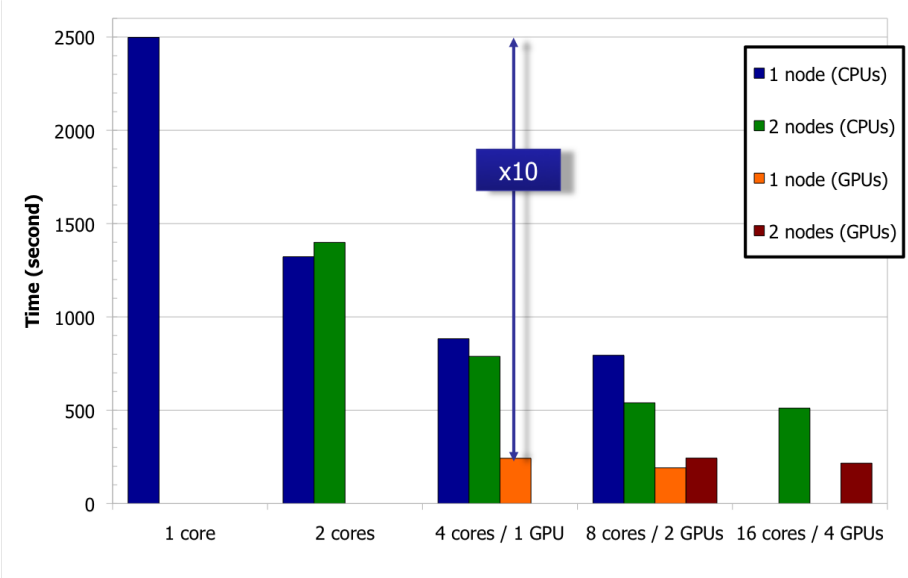

Figure 9: RTM Scaling Performance.