Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 976 – 985

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

# Hands-on Performance Tuning of 3D Finite Difference Earthquake Simulation on GPU Fermi Chipset

Jun Zhou [a,b,*], Didem Unat [c], Dong Ju Choi [a], Clark C. Guest [b], Yifeng Cui [a]

[a] San Diego Supercomputer Center, Univ. of California at San Diego, 10110 Hopkins Drive, La Jolla, CA92093, United States
[b] Dept. of Electrionic and Computer Engineering, Univ. of California at San Diego, 9500 Gilman Drive, La Jolla, CA92093, United States
c Dept. of Computer Science and Engineering, Univ. of California at San Diego, 9500 Gilman Drive, La Jolla, CA92093, United States

**Abstract**

3D simulation of earthquake ground motion is one of the most challenging computational problems in science. The emergence of graphic processing units (GPU) as an effective alternative to traditional general purpose processors has become increasingly capable in terms of accelerating scientific computing research. In this paper, we describe our experiences in porting AWP-ODC, a 3D finite difference seismic wave propagation code, to the latest GPU Fermi chipset. We completely rewrote this Fortran-based 13-point asymmetric stencil computation code in C and MPI-CUDA in order to take advantage of the powerful GPU computing capabilities. Our new CUDA code implemented the asymmetric 3D stencil on Fermi to make the best use of GPU on-chip memory for an aggressive parallel efficiency. Benchmark on NVIDIA Tesla M2090 demonstrated 10x speedup versus the original fully optimized AWP-ODC FORTRAN MPI code running on a single Intel Nehalem 2.4 GHz CPU socket (4 cores/CPU), and 15x speedup versus the same MPI code running on a single AMD Istanbul 2.6GHz CPU socket (6 cores/CPU). Sustained single-GPU performance of 143.8 GFLOPS in single precision is benchmarked for the testing case of 128x128x960 mesh size.

***Keywords:*** Earthquake Simulation, 3D Stencil Computation, Performance Tuning, NVIDIA GPU Fermi, CUDA

## 1. Introduction

Earthquake system science is one of the most challenging computational problems in science, its practical mission is to provide society with a better understanding of earthquake causes and effects, with the goal of reducing the potential for loss of lives and property. A variety of numerical methods have been utilized for earthquake simulation, such as Finite Difference (FD) [1, 2], Finite Element (FE) [2], and Spectral Element [3]. In recent years, earthquake simulations have become more and more data intensive, and demand not only computing capability and accuracy, but also computation efficiency.

AWP-ODC is an abbreviation of Anelastic Wave Propagation by Olsen, Day and Cui, a 3D Finite Difference based earthquake simulation code, originally developed by Kim Bak Olsen [4] and optimized by Y. Cui et al. [5].

* Corresponding author (Jun Zhou). Tel: +001-858-822-6178
Email Address: j4zhou@ucsd.edu


doi:10.1016/j.procs.2012.04.104

AWP-ODC has been extensively used by researchers at Southern California Earthquake Center (SCEC) for large-scale wave propagation simulation, dynamic fault rupture studies and improvement of structural models [4, 6, 7]. The code has achieved "M8" in 2010: a full dynamic simulation of a magnitude-8 earthquake on the southern San Andreas Fault up to 2Hz (see Figure 1). As the largest earthquake simulation ever performed at that time, M8 produced 360 sec of wave propagation, sustaining 220 TeraFLOPS for 24 hours on DOE's Cray XT5 Jaguar machine using 223, 074 CPU cores, and was nominated as ACM Gordon Bell Finalist at SC'10 [5]. Recently, AWP-ODC has also been adapted to the reciprocity-based CyberShake simulation [8]. The CyberShake project targets to create a state-wide (California) physics-based seismic hazard map up to 1Hz, which requires more than four hundred million CPU-core hours in order to complete the planned 5000 sites of SGT calculations in the next 2-3 years [8]. On mainframes provisioned with conventional CPUs, however, each site calculation would take thousands of cores up to two days. The practical plan to accelerate AWP-ODC waveform modeling on CPU-GPU clusters is to dramatically reduce the allocation hours needed for the CyberShake 3.0 calculation, with significantly improved time-to-solutions. Currently, three of the top five supercomputers in world (Tianhe-1A, Nebulae and TSUBAME 2.0) are using hybrid CPU-GPU computing mode, installed with latest NVIDIA Fermi GPUs [9]. Development of AWP-ODC code on Single GPU Fermi Chipset is a key step on the way to adopting these powerful supercomputers for Petascale earthquake simulations.

There have been a few seismic wave propagation applications ported to a GPU computing platform which have achieved considerable speedup. Micikevicius [10] described a GPU parallel approach for the 3D Wave Equation Finite Difference and extended the 3D stencil computation code to 4 GPUs (NVIDIA Tesla 10-serial GPU). Abdelkhalek et al. [11] implemented a 3D Finite Difference GPU-based code for seismic reverse migration calculation and achieved 10x and 30x over the serial code running on a general purpose single CPU core. Komtitsch et al. [12-15] developed Finite Difference, Finite Element and Spectral Elements codes for 3D seismic simulation modeling respectively and ported all of them to GPU Clusters (NVIDIA GeForce 8800 and GTX 280) using the same optimization algorithms, which obtained 12x and 20x speedup. Song et al. [16] accelerated 3D viscoelastic equations in Support Operator Rupture Dynamic (SORD) seismic wave prorogation code on NVIDIA Tesla 1060 GPUs and got a maximum of 15x speedup. Okamota et al. [17-19] presented their multi-GPU seismic wave propagation on TSUBAME supercomputers [9] and achieved 2.2 TFLOPS via 120 GPUs. This paper presents our hands-on performance tuning experience and studies optimization approaches of 13-point asymmetric 3D stencil-based Finite Difference code on Fermi for higher computation efficiency, with more focus on aggressive performance on the latest NVIDIA GPU Fermi chipset. We demonstrate performance evaluation between our fully optimized AWP-ODC Fortran MPI code running on different multi-core CPU systems and the new CUDA code running on different GPU chipsets.

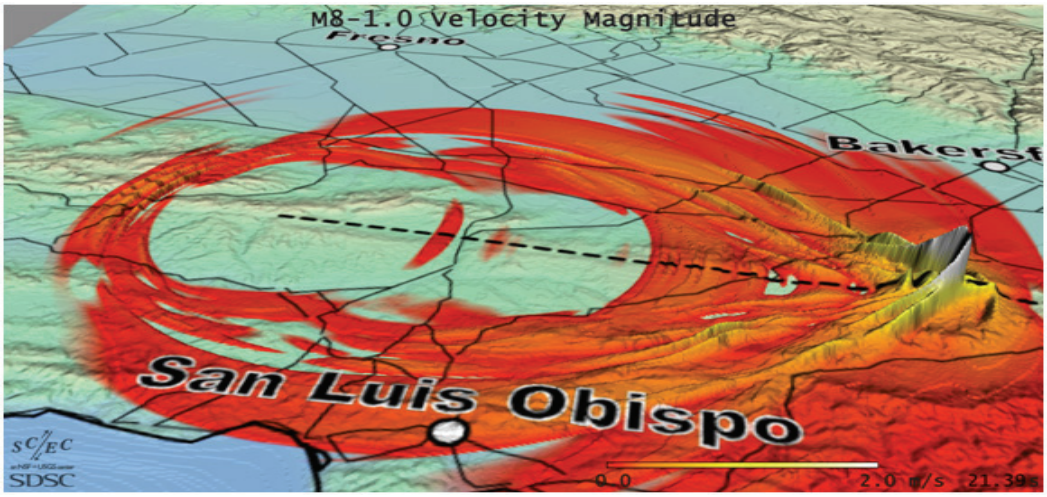



Fig. 1. A visualization image from M8 simulation results by AWP-ODC Fortran MPI code in 2010, which shows ground motion magnitude as a height field, where strongest motion corresponds to the white color and weakest to red. The image shows a cone-like distribution of strong ground motions at the leading edge of the rupture. These so-called mach cones are difficult to observe because they are rare, but important because they are associated with large ground motions, and therefore an essential area of study for large-scale numerical simulations such as a Magnitude 8.0 event [5].

## 2. AWP-ODC Numerical Model

This section describes the formulation of the AWP-ODC numerical model and the analysis of the computation kernels. AWP-ODC solves a 3D velocity-stress wave equation using an explicit method with a staggered-grid finite difference method, fourth-order accurate in space and second-order accurate in time. The coupled system of partial differential equations includes the particle velocity vector $v$ and the symmetric stress tensor $\sigma$ [5]. Suppose:

$$v = (v_x, v_y, v_z) \qquad \sigma = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{bmatrix} \tag{1}$$

Then the governing elastodynamic equations are [5]:

$$\partial_t v = \frac{1}{\rho} \nabla \cdot \sigma \tag{2}$$

$$\partial_t \sigma = \lambda (\nabla \cdot v) \mathrm{I} + \mu (\nabla v + \nabla v^T) \tag{3}$$

$\lambda$ and $\mu$ are the lame coefficient and $\rho$ is the constant density. Simplifying formula (2) and (3) leads to three scalar-valued equations for velocity vector components and six scalar-valued equations for the stress tensor components. Three are listed below, one for velocity vector equation and two for stress equations:

$$\frac{\partial v_x}{\partial t} = \frac{1}{\rho} \left( \frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{yy}}{\partial y} + \frac{\partial \sigma_{zz}}{\partial z} \right) \tag{4}$$

$$\frac{\partial \sigma_{xx}}{\partial t} = (\lambda + 2\mu) \frac{\partial v_x}{\partial x} + \lambda \left( \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} \right) \tag{5}$$

$$\frac{\partial \sigma_{xy}}{\partial t} = \mu \left( \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} \right) \tag{6}$$

AWP-ODC is a memory-intensive application and twenty-one 3D variable arrays are involved in the main computation loop. In addition to the three velocity vector components and six symmetric stress tensor components, 6 temporary variables (r1, r2, r3, r4, r5, r6) and 6 constant coefficients ( $\lambda$ , $\mu$ , $\rho$ , quality factor for S wave $Qs$ and P wave $Qp$ , boundary condition variable Cerjan $Cj$ [20]) are utilized in the numerical modeling. Figure 2 is the pseudo-code of the computation kernels in the main loop for formula (4) and (5):

```
Main Loop:
Do T= timestep 0 to timestep N:
      Compute velocities (vx, vy, vz) using stress (xx, yy, zz, xy, yz, xz)   -   Velocity Computation Kernel
      Update values of velocities (vx, vy) along the surface
      Update values of velocities (vz) based on (vx, vy) along the surface
       Compute stress (xx, yy, zz, xy, xz, yz) based on velocities (vx, vy, vz)   -   Stress Computation Kernel
      Update values of stress (zz, xz, yz) along the surface
END DO

Velocity Computation Kernel (only vx computation is shown here)
vx(i, j, k) += d1(i,j,k)*( c1*(xx(i, j, k) – xx(i-1, j, k)) + c2*(xx(i+1, j, k) – xx(i-2, j, k))
                           c1*(xy(i, j, k) – xy(i, j-1, k)) + c2*(xy(i, j+1, k) – xy(i, j-2, k))
                           c1*(xz(i, j, k) – xz(i, j, k-1)) + c2*(xz(i, j, k+1) – xz(i, j, k-2)) )

Stress Computation Kernel (only xy computation is shown here)
vxy = c1*( vx(i, j+1, k) – vx(i, j, k) ) + c2*( vx(i, j+2, k) – vx(i, j-1, k) )
vyx = c1*( vy(i, j, k) – vy(i-1, j, k) ) + c2*( vy(i+1, j, k) – vy(i-2, j, k) )
xy(i, j, k) = xy(i, j, k) + xmu1(i, j, k)*(vxy+vyx) + x1*r4(i, j, k)
r4(i, j, k) = x2(i, j, k)*r4(i, j, k) + h1(i, j, k)*(vxy + vyx)
```

Fig. 2. AWP-ODC pseudo code for computation kernels: vx stands for $V_x$ . xy stands for $\sigma_{xy}$ . c1, c2 and x1 are scalar constants. r4 is an intermediate variable and also updated in stress calculation during iterations. d1 is the constant density $\rho$ , while xmu1 and x2 are 3D static lame coefficient variables derived from $\lambda$ and $\mu$ .

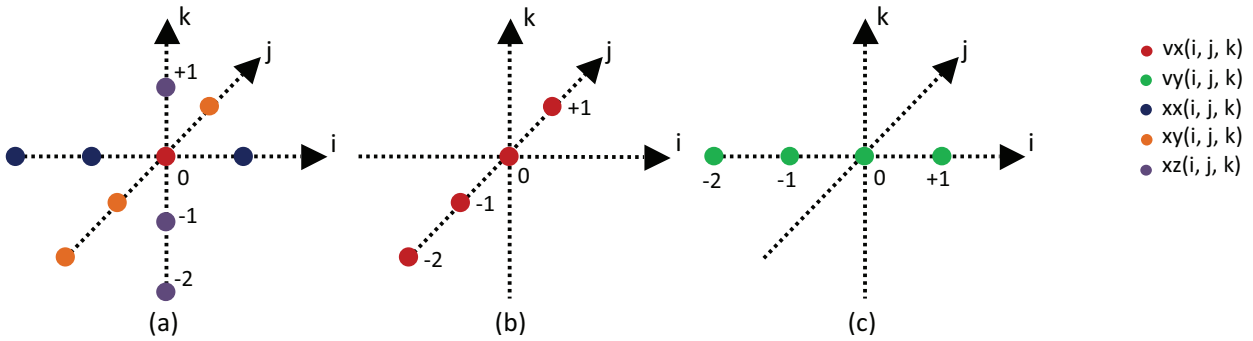



Fig. 3. Analysis of AWP-ODC Computation Kernels based on the pseudo-code in Figure 2: (a) is the velocity computation kernel for vx, 13 point asymmetric stencil computation involving 3 stress components. (b) and (c) are for the stress component xy computation, two asymmetric stencils in x and y directions respectively. Asymmetric stencil computation is the same as the regular stencil computation for single CPU/GPU programming, but the asymmetric property is good for MPI optimization for multi-CPUs or GPUs programming.

The 13 point asymmetric stencil computation for vx in AWP-ODC main loops is shown in Figure 3a, where reads occur 12 times in three different 3D arrays and writes occur only once in one 3D array. Stress component xy calculation includes two 1D asymmetric stencils (Figure 3b and 3c), with only 4 reads from a single 3D array and no writes. Table 1 summarizes the analysis of the three kernels in Figure 2, showing that AWP-ODC is a memory-bound application because of the low FLOPS to bytes ratio (the average operation intensity is around 0.5), which means the application has poor temporal data locality and the performance is dominated by the memory system or arithmetic throughput [21].

Table 1. Analysis of Computation Kernels in AWP-ODC Main Loop.

| Kernels | Reads | Writes | FLOPs | FLOPs/Bytes |
|---|---|---|---|---|
| Velocity Computation Kernel | 51 | 3 | 86 | 0.398 |
| Stress Computation Kernel | 85 | 12 | 221 | 0.569 |
| Total | 136 | 15 | 307 | 0.508 |

## 3. GPU Fermi and CUDA

NVIDIA GPU Fermi is a more advanced GPU computing architecture, its fastest Product is “Tesla M2090”, released in May 2011. Tesla M2090 has 512 CUDA parallel processing cores, and delivers 1331 GFLOPS in single-precision (SP) performance, with 6GB GDDR5 memory size and 177GB/sec memory bandwidth (ECC off) [22]. For Fermi Chipset, each streaming multiprocessor (SM) includes 32 CUDA cores, along with 16 load/store units for memory operations to improve I/O performance, 4 special-function units (SFU) to handle complex math operations and 64k local SRAM split between hardware-managed L1 cache and software-managed shared memory. The local SRAM can be split into two different modes: 16KB/48KB or 48KB/16KB based on the users’ requirement. One of the partitions “shared memory” is the fast memory that can be accessed by all 32 cores in the same SM. Each Fermi Chipset also has a 768KB L2 cache shared by all SMs. In the L2 cache subsystem, the atomic instructions (a set of read-modify-write memory operations) have been improved 5 to 20 times faster than the NVIDIA first generation GPU chipset [23].

CUDA is a general programming language for programming on GPUs and a parallel extension of C/C++ developed by NVIDIA Corporation [24]. CUDA programs are launched on the GPU device as sequences of kernels using a hierarchy of threads. Threads are grouped into blocks and thread blocks run on SMs in groups of 32 threads. The collection of the 32 threads running on same SM is referred as a warp, and all threads in the same warp execute same instructions simultaneously. For CUDA programs, each thread has its own private registers, and threads within the same thread block share the same on-chip fast shared memory. Threads also have to access global memory for data, which is off-chip device memory with high access latency (almost 500 clock cycles). To achieve high performance, CUDA program design must focus on minimizing off-chip memory access and maximizing computing parallel efficiency.

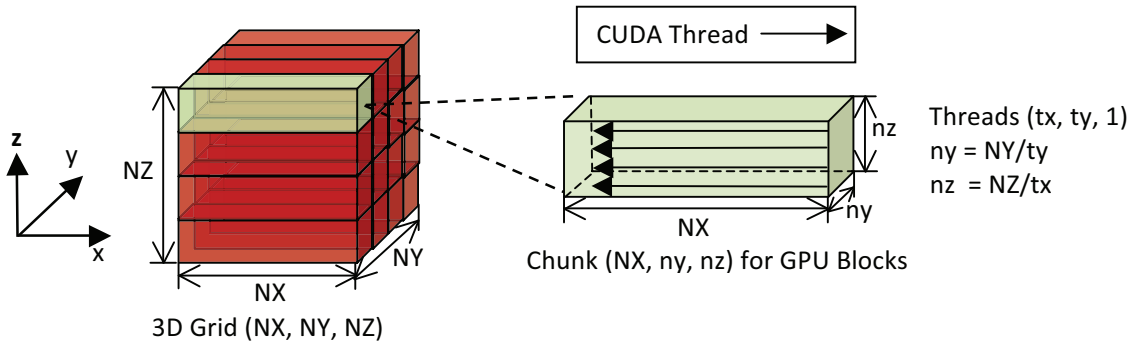



Fig. 4. 2D Domain Decomposition for GPU Kernels: the 3D Grid (NX, NY, NZ) is decomposed only in y and z directions. Suppose each block has (tx, ty, 1) threads, then the kernel function has (NZ/tx, NY/ty, 1) blocks and the chunk size for each block will be (NX, ny, nz).

## 4. Implementation and Performance Tuning

This section describes the AWP-ODC CUDA implementation and performance tuning strategies on a single NVIDIA GPU M2090, the Fermi Architecture with 2.0 capability. The GPU codes discussed in this section are compiled by the CUDA compiler nvcc 4.0, with options: *-O4 -Xptxas -dlcm=ca -maxrregcount=63 -use_fast_math -arch=sm_20*. Two extra layers are added to each direction (called ghost cells) in the kernel stencil computation, which means the actual size for the 3D grid in memory is (NX+4, NY+4, NZ+4) instead of (NX, NY, NZ).

Prior to the computation, the CPU initializes and allocates all twenty-one 3D variable arrays from input files or parameters, then copies all related 3D arrays into the GPU device memory via the PCI 2.0 Express bus and executes the velocity and stress kernels as shown in Figure 2. Upon the completion of the main loop, result data are transferred back from GPU to CPU main memory for output. Full utilization of on-chip memory is the key to achieve high performance for GPU-based applications. Our optimization mechanisms can be summarized into five steps to achieve this goal:

*Step 1: Read-Only Memory Cache*: Texture and Constant memory in the GPU device has its own cache. As discussed in Section 2, six 3D variable arrays are constant coefficients and we put them into the texture memory to take advantage of the texture cache. All scalar constants in Figure 2 are put into the constant memory to save register use. All other 15 arrays have to be stored in global memory because their values are updated during iteration.

*Step 2: 2D Domain Decomposition*: CUDA is an extended language of C/C++, so memory storage for 3D arrays will be fast in the z direction and slow in the x direction. To obtain a better cache hit rate and allow all threads in the same wrap to access data along the fast z axis instead of the slow x axis, we decompose the 3D Grid only in y and z directions. Each thread will calculate the entire NX for a given 2D (y, z) location as shown in Figure 4.

Table 2. Performance comparison between two implementations with different 2D decomposition geometry derived from a baseline code, which is based on direct global memory access without any optimization. Benchmark runs for 800 timestep for time-to-solution measurement. The thread block for GPU Kernels is (64, 8, 1), so the value B_x = 64 and B_y=8 in the table.

| **3D Grid Size** (NX x NY x NZ) | **CUDA CODE for (x, y) 2D Decomposition**<br>1. x = blockIdx.x*B_x + threadIdx.x + 2;<br>2. y = blockIdy.y*B_y + threadIdy.y + 2;<br>3. for (z = NZ+1; z>=2; --z)<br>4. Velocity and Stress Computation | **CUDA CODE for (y, z) 2D Decomposition**<br>1. z = blockIdx.x*B_x + threadIdx.x + 2;<br>2. y = blockIdy.y*B_y + threadIdy.y + 2;<br>3. for (x = NX+1; x>=2; --x)<br>4. Velocity and Stress Computation |
|---|---|---|
| 128x128x128 | 0.242 sec/per timestep | 0.008 sec/per timestep |
| 128x128x256 | 0.490 sec/per timestep | 0.020 sec/per timestep |
| 128x256x256 | 0.977 sec/per timestep | 0.040 sec/per timestep |
| 256x256x256 | 1.956 sec/per timestep | 0.086 sec/per timestep |

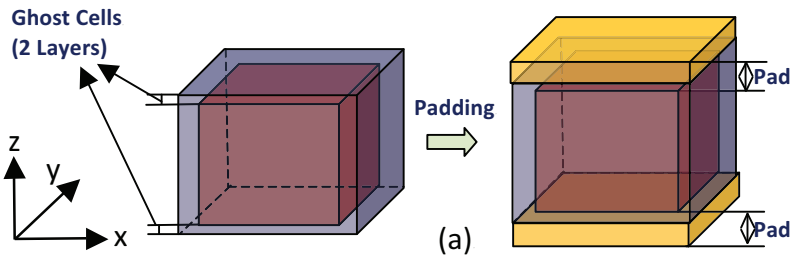
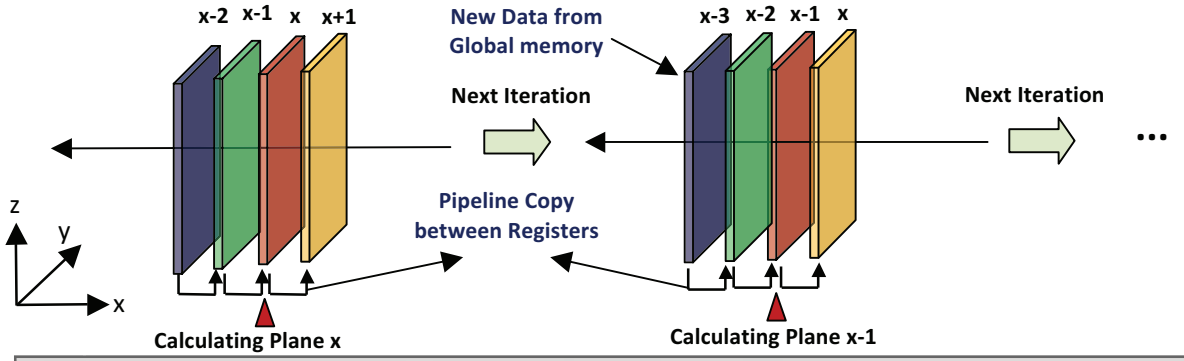



Fig. 5. (a) 3D array padding: red cube represents original 3D grid, blue cube represents ghost cells with 2 extra planes in each plane. The yellow part is the padding arrays, including the 2 layer ghost cells in z axis. (b) is CUDA code for step 2&3, fast z for better cache hit rate after padding.

Since the thread block is also a 3 dimension topology and fast in the x direction, threads in the x direction must correspond to the 3D Grid z direction, while threads in the y direction correspond to the 3D Grid y direction. Since 32 threads in a wrap are executed in Fermi at the same time, the number of threads in the x direction is considered to be a multiple of 32 for better performance. Table 2 is the performance comparison between 2D decomposition in (x, y) direction and (y, z) direction, showing that the code performance for decomposition in (y, z) direction is 25x faster than in (x, y) direction.

*Step 3: Global Memory Coalesced:* For maximum performance, memory access must be coalesced as were accesses to global memory [25]. CUDA provides some functions such as "cudaMallocPitch" and "cudaPitchedPtr" to help ensure aligned memory access. However, as shown in Figure 5a, instead of using these library functions, we manually pad zeros onto the boundaries in the z axis of the 3D grid to align memory to the inner region. The padding size plus NZ should also be a multiple of 32, and the two layers of ghost cells are included in the padding.

*Step 4: Register Optimization:* Private registers are the fastest on-chip GPU memory and take only one clock cycle for data access or calculation. Neighboring data in y and z directions are close to each other in the aligned memory, therefore these data can be cached or prefetched into shared memory. For neighboring data in our travel direction (x axis), three out of four values can be reused for computation on the next iteration. We can define four private registers to store these values and utilize pipeline copy between registers, reducing 75% of the global memory access in x direction. This algorithm has been widely used in seismic 3D finite difference [10-19], and Figure 6 illustrates how it works in our CUDA code.




**VX computation kernel in thread (ty, tz) is to describe the Register Optimization:**

Four registers: Y (Yellow), R (Red), G (Green), B (Blue)
Global memory: vx[i, ty, tz], xx[i, ty, tz] - 3D array (NX+4)*(NY+4)*(NZ+2*pad)
On-chip memory: xy[i, j, k], xz[i, j, k] - One plane Loaded to L1 cached or pre-fetched to shared memory

1. Preload R=xx[NX+2,ty,tz], G=xx[NX+1, ty, tz], B=xx[NX, ty, tz]
2. For (i=NX+1; i>=2; i--)
   a. Y=R; R=G; G=B; - pipeline copy between registers, very fast
   b. B = xx[i-2, ty tz]; - only access xx from global memory once in each iteration
   c. VX Computation via Y, R, G, B, xy & xz
3. End For Loop and Return vx

Fig. 6. Illustration of register optimization: pipeline register copy to reduce global memory access for asymmetric 13-points stencil computation.

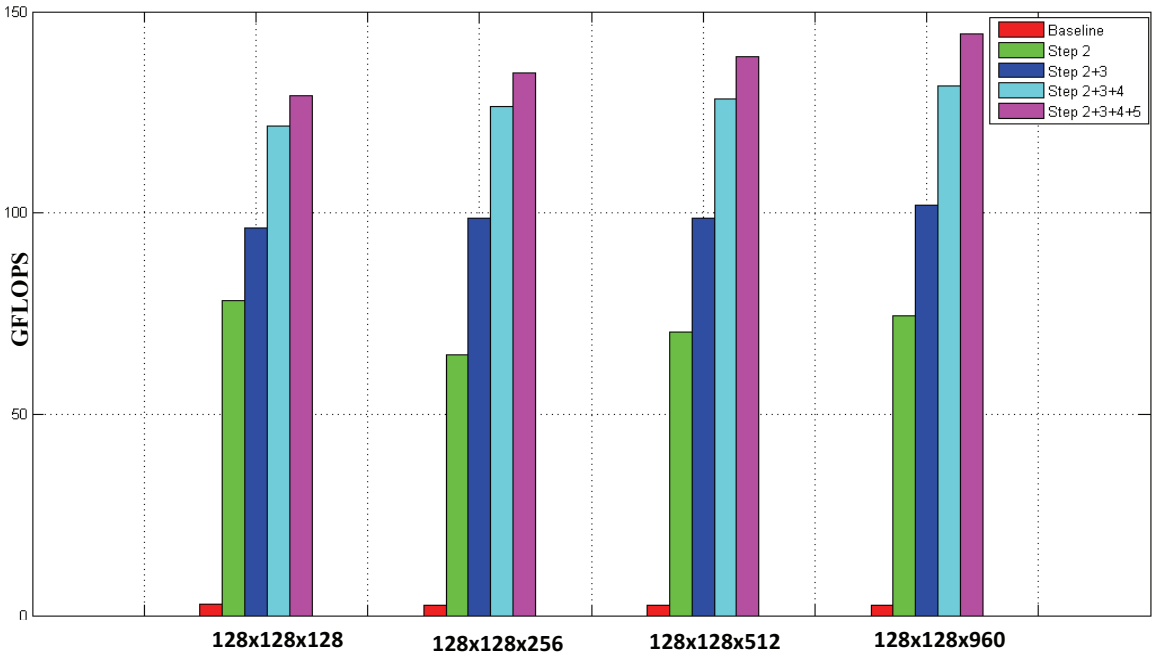



Fig. 7. Increment of computing GFLOPs achieved by optimization steps for various 3D grid sizes tested on NVIDIA M2090. The baseline version is based on pure global memory implementation without any optimization, and other versions are optimized step by step.

*Step 5: L1 Cache vs Shared Memory:* GPU Fermi provides 64KB on-chip memory and two configuration modes to programmers: 48KB L1 cache and 16KB shared memory for *preferL1*, or 16KB L1 and 48KB shared memory for *prefershared.* We chose preferL1 over perfershared in our AWP-ODC CUDA implementation because the two layer ghost cells are the bottleneck for high efficiency when fetching data into shared memory. Suppose one GPU block is to compute a plane with the size16x32, then the grid size to be fetched into shared memory should be 20x36 due to the 2 layer ghost cells. Since each block only has 512 threads, some extra data loading operations are required to load data in ghost cells, thus load balance issues might occur since not all threads in the block are participating in ghost cell loading, and threads synchronization has to be finished before any computation. We also implemented a shared memory version based on step 4, which is to prefetch one (y, z) plane to shared memory instead of L1 cache in Figure 6, and choose *preferShared* over *preferL1* for on-chip memory configuration. However, the time to solution for benchmark size 128x128x960 between the shared memory version and the code implemented in step 4 are almost the same (0.037 sec/per timestep v.s. 0.038 sec/per timestep), which means shared memory obtain little performance benefit after the first four optimization steps in the AWP-ODC application. Shimokawabe [26] chose similar *preferL1* over *preferShared* for a similar stencil-based phase-field simulation for dendritic solication on Tesla M2050.

Figure 7 shows the increment of computing FLOPs achieved by these optimization steps. The computing performance of our code for size 128x128x960 on Tesla M2090 is 143.8GFLOPs for single precision. We consider this outstanding performance for a memory-bound stencil computation. We compare this performance with Schafer [27] achievement of 152.2GFLOPs for single precision with a classical 3D Jacobi Solver. While Schafer's test was on a slower Fermi device (Tesla C2050), but our code is more memory intensive than the 3D Jacobi Solver.

## 5. Experiments and Performance Comparisons

The new AWP-ODC CUDA implementation has been carefully validated for correctness. We verify the CUDA code with original AWP-ODC Fortran MPI code for production simulations for comparision, based on point source propagated across the whole 3D mesh (128x128x128). Figure 8 is the first 500 timesteps of the log value of the velocity vx recorded at the source point between two codes, showing that the results are almost identical except neglectable roundoff difference caused by different programming languages and compilers.

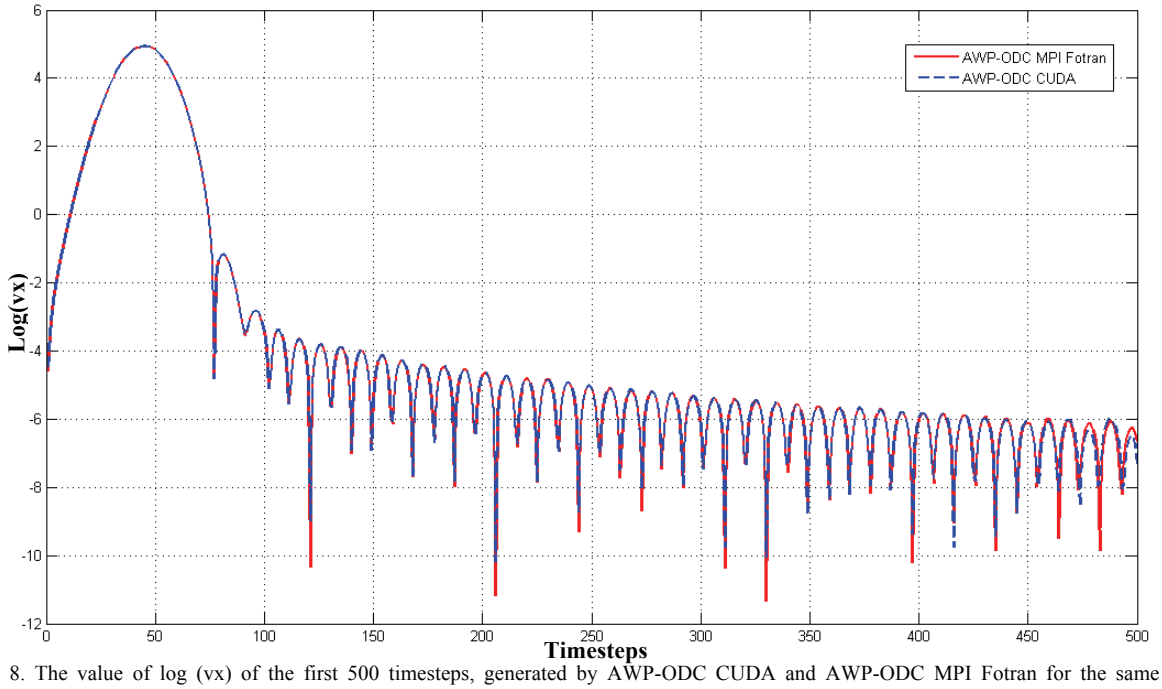



Fig. 8. The value of log (vx) of the first 500 timesteps, generated by AWP-ODC CUDA and AWP-ODC MPI Fotran for the same wave propagation earthquake simulation, is compared for validation. The 3D grid size for the earthquake simulation is 128x128x128, and the single source point has 91 timestep inputs.

To compare the performance between the AWP-ODC CUDA code and the original fully optimized AWP-ODC Fortran MPI code (2010 Gordon Bell Finalist [5]), we chose two multi-core CPU clusters and three NVIDIA GPU devices. First multi-core CPU cluster is the TRITON machine located at the San Diego Super Computer Center (SDSC) [28]. Second multi-core CPU cluster is the KRAKEN machine located at the National Institute for Computational Sciences (NICS) [9]. The three GPU devices are NVIDIA Tesla C1060, Tesla C2050 and Tesla M2090, and C1060 is the NVIDIA first generation GPU, while C2050 and M2090 belongs to GPU Fermi. Table 2 is the summary of the hardware characteristics of all testbeds.

Table 2. Hardware characteristics of all testbeds: two multi-core CPU clusters and three GPU devices

| CPU/GPU | Concurrency | Frequency | L1 Cache/Shared Memory | Memory Size | Memory Bandwidth |
|---|---|---|---|---|---|
| Intel Nehalem 5530 on TRITION | 2 threads x 4 cores | 2.4 GHz | 32 KB L1 | 3 GB/core | 25.6 GB/s |
| AMD Istanbul on KRAKEN | 6 cores | 2.6 GHz | 64 KB L1 | 1.33 GB/core | 25.6 GB/s |
| NVIDIA C1060 | 240 cores | 1.3 GHz | 16 KB Shared Memory | 4 GB in total | 102.4 GB/s |
| NVIDIA C2050 | 448 cores | 1.15 GHz | 64 KB L1/Shared Memory | 3 GB in total | 144 GB/s |
| NVIDIA M2090 | 512 cores | 1.3 GHz | 64 KB L1/Shared Memory | 6 GB in total | 177 GB/s |

Figure 9 is the performance comparsion between AWP-ODC CUDA code running on the three GPU devices and AWP-ODC Fortran MPI code running on the two multi-core CPU clusters. All benchmark experiments run for 800 timesteps and the measurement is focused on the averge time per timestep. For GPU devices, The NVDIA Tesla C1060 is 1.3 capable and the compiler is NVCC 4.0 with options "-O4 -Xptxas -dlcm=ca -maxrregcount=63 -use_fast_math -arch=sm_13". The AWP-ODC CUDA code for Tesla C1060 is a little different from the one implemented in Section 4, and we used shared memory for Step 5 as there is no L1 cache in Tesla C1060. The Tesla C2050 and M2090 are 2.0 capable using the same NVCC 4.0 compiler and the compiler options except "-arch=sm_20". For multi-core clusters, we only use single socket to run the AWP-ODC Fortran MPI code, which means four MPI threads on a single socket of TRITON machine and six MPI threads on a single socket of KRAKEN machine. The compiler is Portland Group Compiler (PGI) mpif90 v10.5 with compiler options "-O3 -fastee -Mflushz -Mdaz -Mnontemporal -Mextend and -Mfixed".

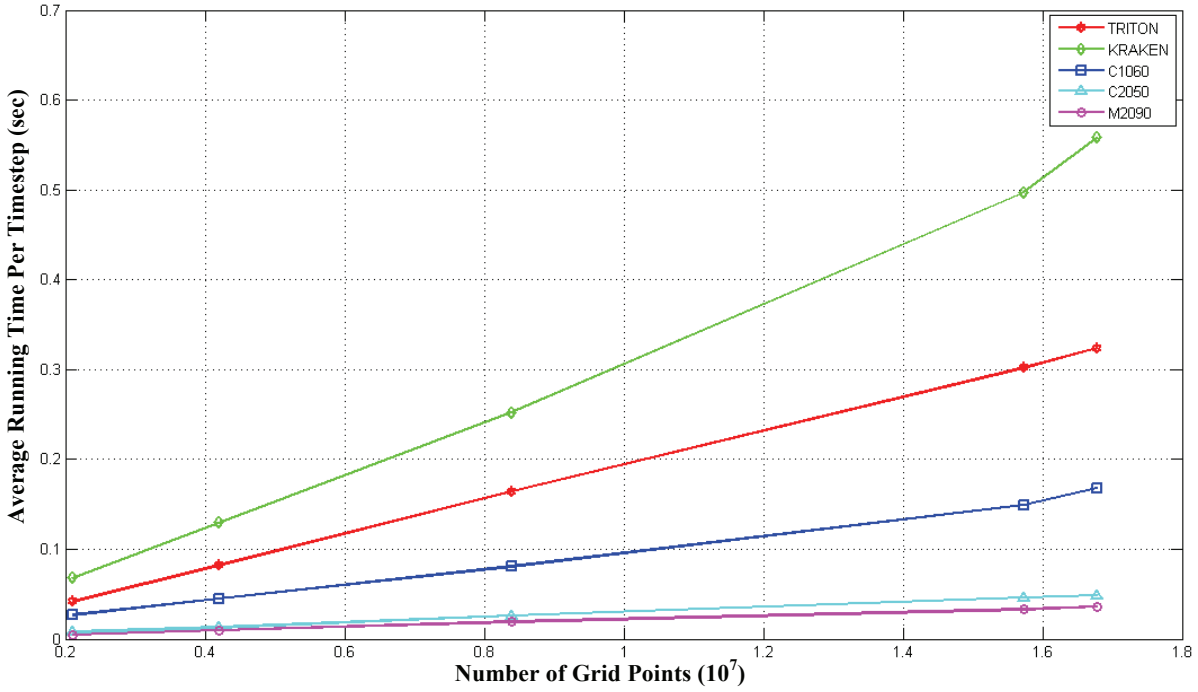



Fig. 9. Performance comparison between AWP-ODC CUDA code running on GPU devices and AWP-ODC Fotran MPI code running on Multi-core CPU clusters with different benchmark sizes.

In Figure 9, for the same AWP-ODC CUDA code, the average running time per timestep on M2090 is almost 30% less than on C2050, this is because M2090 has two more SMs and higher clock speed. The shared-memory based code running on Tesla C1060 is 4.5x slower than running on M2090, due to the fewer CUDA cores, lack of L1 cache, and lower memory bandwidth. Note that the CUDA code running on M2090 is almost 10x faster than the Fortran MPI running on single Intel Nehalem 2.4GHz CPU on TRITON with four MPI threads, and 15x faster than the MPI code running on AMD Istanbul 2.6GHz CPU on KRAKEN with six MPI threads.

## 6. Conclusion and Future Plans

We presented implementation and hands-on optimization tuning of a 3D finite difference earthquake simulation code on the GPU Fermi Chipset. The performance of this memory intensive stencil computation code has been significantly improved through effective utilization of GPU on-chip memory. The single NVIDIA Tesla M2090 GPU benchmark demonstrates sustained performance of 143.8 GFLOPS in single precision for 128x128x960 mesh size, at approximately 10% of the peak system performance, 10x faster compared to the MPI CPU code running on Intel Nehalem 2.4 GHz CPU socket with 4 thread-cores and 15x speedup from running on AMD Istanbul 2.6GHz CPU with 6 thread-cores. To our knowledge this is the highest single-GPU performance measured from a seismic application. Our MPI-CUDA code is being tested on the XSEDE resource Keeneland [9], strong scalability on multi-GPUs will be addressed in another paper to come, where emphasis will be on re-engineering of multi-GPU optimization. We also plan to add the advanced MPI-IO to handle data-intensive volumetric I/O and port the CUDA code in coming months to the next generation GPU architecture “Kepler” Titan at Oak Ridge National Lab [29]. Our ultimate goal is to establish a new simulation capability for effective use of petascale CPU-GPU clusters, with improved efficiency and computing flops performance on challenging heterogeneous systems. The MPI-CUDA code will be adapted to run large number of Cybershake simulations with efficient scientific workflows [30] incorporated for petascale earthquake simulations.

## References

1. K. B. Olsen, R.J. Archuleta. Three-dimensional Simulation of Earthquakes on the Los Angeles Fault System. Bulletin of the Seismological Society of America June 1996, vol. 86, no. 3, pp.575-596.

2. K. J. Marfurt. Accuracy of finite-difference and finite-element modeling of the scalar and elastic wave equations. Geophysics 49, 533, 1984.

3. D. Komatitsch and J. P. Vilotte. The spectral element method: An efficient tool to simulate the seismic response of 2D and 3D geological structures. Bulletin of the Seismological Society of America. April 1998, vol. 88, no. 2, pp. 368-392.

4. K. B. Olsen, Simulation of Three-Dimension Wave Propagation in the Salt Lake Basin. Doctoral Dissertation, University of Utah, 1994.

5. Y. Cui, K. B. Olsen, T.H.Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. Panda, A. Chourasia, J. Levesque, S. M. Day, and P. Maechling. Scalable Earthquake Simulation on Petascale Supercomputers. In Proceedings of the 2010 ACM/IEEE conference on Supercomputing, SC'10, pp. 1-20, Nov. 2010.

6. K. B. Olsen, S. M. Day, J. B. Minister, Y. Cui, A. Chourasia, M. Faerman, R. Moore, P. Maechling and T. H. Jordan. Strong Shaking in Los Angles Expected from Southern San Andreas Earthquake. In Geophysical Research Letters. Vol. 33. 2006.

7. L. Dalguer and S. M. Day. Staggered-grid Split-node Method for Spontaneous Rupture Simulation. In Journal of Geophysical Research. Vol. 12, 2007.

8. http://*scec.usc.edu/scecpedia/CyberShake*

9. http://www.top500.org/lists

10. P. Micikevicius. 3D Finite-difference Computation on GPUs Using CUDA. In GPGPU-2: Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units, Washington, DC, USA, 2009, pp. 79-84.

11. R. Abdelkhalek, H. Calandra, O. Coulaud, J. Roman, G. Latu. Fast Seismic Modeling and Reverse Time Migration on a GPU Cluster. Internaltion Conference on High Performance Computing and Simulation, 2009, HPCS'09. Leipzig, Germany. pp. 36-43. Aug. 07, 2009.

12. D. Komatitsch, D. Michea, G. Erlebacher. Porting a high-order Finite-element Earthquake Modeling Application to NVIDIA Graphic Cards Using CUDA. J. Parallel Distributed Comput. 69 (5) (2009) pp.451-460.

13. D. Komatitsch, G. Erlebacher, D.Goddeke, D. Michea. High-order Finite-element Sesimic Wave Propagation Modeling with MPI on a Large GPU Cluster. J. Comput. Phys. Vol. 229, Issue 20, Oct. 2010, pp.7692-7714.

14. D. Komatitsch, D. Goddeke, G. Erlebacher, D. Michea. Modeling the Propagation of Elastic Waves Using Spectral Elements on a Cluster of 192 GPUs. Computer Science – Research and Development. Vol. 25, No. 1-2. pp. 75-82.

15. D. Michea, D. Komatitisch. Accelerating a 3D finite-difference Wave Propogation Code Using GPU Grapics Cards. Geophys. J. Int. 182(1) (2010), pp. 389-402

16. S. Song, T. Dong, Y. Zhou, D. Yuan, and Z. Lu. Sesmic Wave Propagation Simulation Using Support Operator Method on Multi-GPU System. In Technical Report, University of Minnesota, 2010.

17. T. Okamoto, H. Takenaka, T. Nakamura, and T. Aoki. Accelerating large-scale simulation of seismic wave propagation by multi-GPUs and three-dimensional domain decomposition, Earth Planets and Space, 62, pp.939-942, 2010

18. T. Okamoto, H. Takenaka, T. Nakamura, and T. Aoki. Accelerating Simulation of Seismic Wave Propagation by Multi-GPUs. AGU 2010 Meeting, San Francisco, California, Dec. 13-17, 2010.

19. T. Okamoto, H. Takenaka, T. Hara, T. Nakamura, and T. Aoki. Rupture Process And Waveform Modeling of The 2011 Tohoku-Oki, Magnitude-9 Earthquake. AGU 2011, San Francisco, California, Dec. 5-9, 2011.

20. Simone, A., and S. Hestholm, Instabilities in applying absorbing boundaryconditions to high-order seismic modeling algorithms, Geophysics, 63,1017– 1023, 1998.

21. S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. Commun. ACM, 52:65-76. April 2009.

22. NVIDIA Cooperation. http://developer.nvidia.com/content/tesla-m2090-announced

23. P. N. Glaskowsky. NVIDIA's Fermi: The First Complete GPU Computing Architecture. NVIDIA White Paper, 2009.

24. J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. In SIGGRAPH'08: ACM SIGGRAPH 2008 classes. Pp. 1-14, ACM, 2008.

25. NVIDIA C Programming Guide, Version 4.0. NVIDIA Cooperation. May, 2011.

26. T. Shimokawabe, T. Aoki, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, A. Nukuda, S. Matsuoka. Peta-scale Phase-field Simulation for Dendritic Solidification on the TSUBAME 2.0 Supercomputer. In Proceeding of International Conference for High Performance Computing, Networking, Storage and Analysis. SC'11, 2011.

27. A. Schafer, D. Fey. High Performance Stencil Code Algorithms for GPGPUs. In Proceeding of International Conference on Computational Science, pp.2027-2036. ICCS 2011.

28. Triton Machine. http://tritonresource.sdsc.edu/

29. K. Bent. NVIDIA's Tesla GPU to Fuel World's Fastest Supercomputers. CRN Tech News. http://www.crn.com/news/components-peripherals/231900675/nvidia-8217-s-tesla-gpus-to-fuel-world-8217-s-fastest-supercomputer.htm.

30. J. Zhou, Y. Cui, S. Davis, C. C. Guest and P. Maechling. Workflow-Based High Performance Data Transfer and Ingestion to Petascale Simulations on TeraGrid.Computational Science and Optimization (CSO'10), 2010 Third Intl. Joint Conference on, vol.1, pp.343-347, May 2010.