Accelerating DynEarthSol3D on Tightly Coupled CPU-GPU Heterogeneous Processors

Tuan Ta[a,*], Kyoshin Choo[a,*], Eh Tan[b,*], Byunghyun Jang[a,**], Eunseo Choi[c,**]

[a] *Department of Computer and Information Science, The Unversity of Mississippi, University, MS, USA.*
[b] *Institute of Earth Sciences, Academia Sinica, Taipei, Taiwan.*
[c] *Center for Earthquake Research and Information, The University of Memphis, Memphis, TN, USA.*

Abstract

DynEarthSol3D (Dynamic Earth Solver in three dimensions) is a flexible, open-source finite element solver that models the momentum balance and the heat transfer of elasto-visco-plastic material in the Lagrangian form using unstructured meshes. It provides a platform for the study of the long-term deformation of earth's lithosphere and various problems in civil and geotechnical engineering. However, the continuous computation and update of a very large mesh poses an intolerably high computational burden to developers and users in practice. For example, simulating a 2-million-element mesh for 1,000 time units takes 1.4 hours on high-end desktop CPU. In this paper, we explore tightly coupled CPU-GPU heterogeneous processors to address the computing concern by leveraging their new features and developing hardware architecture aware optimizations. Our proposed key optimization techniques are three-fold: memory access pattern improvement, data transfer elimination and kernel launch overhead minimization. Experimental results show that our proposed implementation on a tightly coupled heterogeneous processor outperforms all other alternatives including traditional discrete GPU, quad-core CPU using OpenMP, and serial implementations by 67%, 50%, and 154% respectively even though the embedded GPU has significantly less number of cores than high-end discrete GPU.

Keywords: Computational Tectonic Modeling, Long-term Lithospheric Deformation, Heterogeneous Computing, GPGPU, Parallel Computing

1. Introduction

The combination of an explicit finite element method, the Lagrangian description of motion, and the elasto-visco-plastic material model has been implemented in a family of

*Principal corresponding author
**Corresponding author
Email addresses: tqta@go.olemiss.edu (Tuan Ta), kchoo@go.olemiss.edu (Kyoshin Choo), tan2@earth.sinica.edu.tw (Eh Tan), bjang@cs.olemiss.edu (Byunghyun Jang), echoi2@memphis.edu (Eunseo Choi)

codes following the Fast Lagrangian Analysis of Continua (FLAC) algorithm [10]. These specific implementations of the generic FLAC algorithm have a track record of applications that demonstrate the method's aptitude for Long-term Tectonic Modeling (LTM) (e.g., [4, 6, 7, 12, 14, 16, 19, 20]). The original FLAC requires a structured quadrilateral mesh which severely limits the meshing flexibility, one of the major advantages of finite element method. Flexibility in meshing is all the more important for LTM in which strain localization needs to be adequately captured by a locally refining a mesh, which is challenging for a structured mesh. Additionally, each quadrilateral is decomposed into two sets of overlapping linear triangles that guarantee a symmetrical response to loading but leads to redundant computations. On the other hand, FLAC uses an explicit scheme for the time integration of the momentum equation in the dynamic form as well as for the constitutive update, making it relatively easy to implement complicated constitutive models.

By critically evaluating the strengths and weaknesses of the FLAC algorithm, Choi et al.[8] created a new code, DynEarthSol2D, and Tan et al.[25] further extended it to three dimensions, DynEarthSol3D. DynEarthSol3D (**Dyn**amic **Earth Sol**ver in three dimensions) is a robust, flexible, open source finite element code for modeling non-linear responses of continuous media and thus suitable for LTM. DynEarthSol3D written in standard C++ is multi-threaded and freely distributed through a public repository[1] under the terms of the MIT/X Windows System license. DynEarthSol3D inherits desirable features of FLAC such as explicit schemes while modernizing it at the same time. The most notable improvement is the removal of the restrictions on meshing. As a result, one can solve problems on unstructured triangular or tetrahedral meshes while keeping the simple explicit constitutive update that made FLAC attractive in LTM. The use of unstructured mesh enables adaptive mesh refinement in regions of highly localized deformation and discretization of domain geometries that are challenging to discretize into a structured mesh. However, sequential mesh computations and updates are very compute-intensive, resulting in poor performance in practice. To process a large mesh composed of 2 million elements, it takes 1.4 hours for serial implementation to finish 1,000 time steps on a high-end CPU. This huge amount of running time places a limitation on both mesh size and resolution.

GPUs (Graphics Processing Units) have been the platform of choice for compute- and data-intensive applications in many computing domains in recent years. GPU-powered computing provides a number of unique benefits that could not be found in any traditional parallel machines such as supercomputers and workstations. This revolutionary computing paradigm of offloading and accelerating compute- and data-intensive portion of applications on GPUs is termed as GPGPU (General-Purpose Computation on GPUs) or GPU computing. When well optimized for target GPU hardware architecture, application performance can be boosted by up to several orders of magnitude.

GPGPU platforms are typically powered by high-end discrete GPUs (i.e., separate graphics card connected through PCI express). While this type of hardware configuration provides best GPU processing power, data transfer overhead associated with physical separation of GPU device and memory from host CPU diminishes the performance gain obtained by GPU

[1] `http://bitbucket.org/tan2/dynearthsol3d`

acceleration. Deteriorated by kernel launch time, such overheads can be a deal breaker. If an application has multiple sections of CPU and GPU computations that are interleaved and data-dependent of each other, repetitive data transfers between host and device are inevitable, and overall application performance is limited by the overhead associated with these transfers. This problem that is not uncommon in many scientific and engineering applications hinders the adoption of GPGPU.

Recent trend in microprocessor industry is that CPU and GPU are fabricated on a single die sharing a memory system at a cache level [3, 13]. Such tightly coupled CPU-GPU heterogeneous processors provide solutions to several limitations of traditional discrete GPUs such as aforementioned data transfer overhead [9, 24], limited GPU device memory (i.e., GDDR) size [24], and disjoint memory address space. CPU and GPU on tightly coupled heterogeneous processors share the same data in a unified physical memory (data transfer is no longer needed) and also the unified system memory is typically a lot larger (e.g. 32 GB) than discrete GPU device memory (e.g. 4GB).

In this paper, we present the acceleration of DynEarthSol3D on tightly coupled CPU-GPU heterogeneous processors, leveraging a new unified memory architecture to eliminate data transfer overhead. We also revisit and address classical GPGPU challenges such as inefficient memory access patterns and frequent kernel launch overhead. The contributions of our paper are summarized below.

- We demonstrate that tightly coupled CPU-GPU heterogeneous processors outperform discrete GPUs by eliminating data transfer overhead, a serious performance bottleneck of DynEarthSol3D on conventional discrete GPUs. This result is encouraging because the computing power of embedded GPU on heterogeneous processor is less than one fourth of that of discrete GPU that we have tested.

- We propose to improve GPU memory performance by changing memory access patterns through data transformation. By restructuring the mesh, high latency random memory access patterns of DynEarthSol3D turn to regular patterns that GPU hardware can handle much more efficiently. As a result, it boosts the performance of GPU kernel execution significantly.

- We propose to merge kernels whenever possible to minimize kernel launch overhead. Intensive data flow and dependency analysis are conducted to identify all kernels that can be merged without causing any correctness issue. As kernels are called repeatedly throughout program execution, total kernel launch overhead is significantly reduced.

- We conduct thorough performance analysis and comparison with other available alternatives: discrete GPU, multi-core CPU using OpenMP, and a serial implementation as baselines.

The rest of the paper is structured as follows. Section 2 describes the computations of DynEarthSol3D and existing problems in its serial implementation on CPU. Section 3 provides the background on GPGPU with explanation of both traditional discrete GPUs and tightly coupled heterogeneous processors. In Section 4, we present our implementation of DynEarthSol3D while focusing on three key optimization techniques. Lastly, Section 5 shows and discusses our experimental results through detailed evaluation of each optimization technique.

2. Computational Flow of DynEarthSol3D

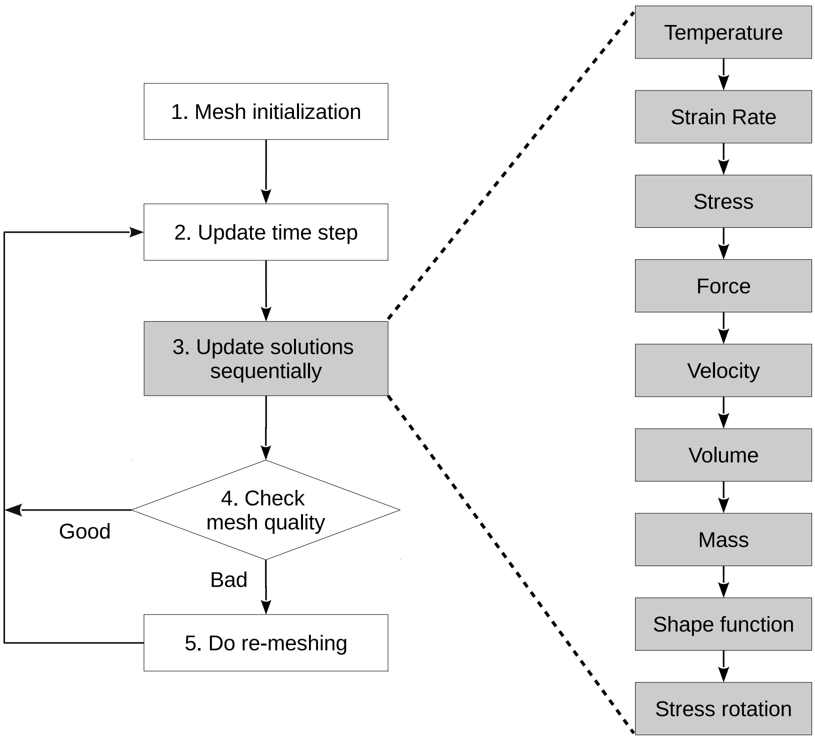


Figure 1: The computational flow of DynEarthSol3D.

Figure 1 visualizes the computational flow of DynEarthSol3D. First, a mesh composed of tetrahedral elements is created by an external mesh generator named *Tetgen* [22]. Each element of the mesh consists of four nodes which function as interpolation points for unknown variables such as velocity and temperature. The program runs through a predefined number of time steps to reach a target model time, which is in LTM typically millions of years to tens of millions of year. In each time step, the temperature field is first updated according to the heat diffusion equation. The updated temperature may be used for computing temperature-dependent constitutive models. Next, based on the current coordinates of nodes and the velocity field, element volume and strain rates are computed. Strain, strain rates and temperature are used to update stress according to an assumed constitutive model. Net acceleration of each node is computed as the net force divided by inertial mass, and the next force is the sum of external body force and internal force arising from the updated stresses. The net acceleration is time-integrated to velocity and displacement. Once the displacement is updated, the coordinates of the nodes are updated. At this stage, the program checks if accumulated deformation has distorted the mesh too severely. If so, Tetgen again is used to regenerate a mesh, each element of which satisfies a certain quality criterion. During this remeshing process, new nodes might be inserted into the mesh, or the mesh topology might change through edge-flipping. This type of remeshing has been proposed as a way of solving large deformation problems in the Lagrangian framework [5]. After the new mesh is created, the boundary conditions, derivatives of shape function, and mass matrix have to be re-calculated. Then, the next time step is initiated unless the current one is the last step.

DynEarthSol3D has options to run either serially or in parallel on multi-core CPU with OpenMP. In the serial version, the elements of the mesh, and the nodes associated with each element, are processed sequentially. In the OpenMP version, when running with P number of threads, in order to prevent race condition among threads, mesh elements are grouped into $2P$ sets corresponding to roughly uniform $2P$ intervals in the x coordinates of their barycenters. Elements in set i are guaranteed to have no common nodes with elements in set $i+2$. To process all elements, elements in set 0, 2, ..., $2(P-1)$ are first processed in parallel, with each set covered by a thread. Then, after all threads finish processing, elements in set 1, 3, ..., $2P-1$ are processed in the same labor division. A good multi-core scaling can be safely achieved this way. When we evaluate the performance of our implementation in this paper, both serial and OpenMP-based implementations are used as baselines.

3. General-Purpose Computing on GPUs (GPGPU)

GPUs provide an unprecedented level of computing power per dollar and energy by running massive number of threads in a Single Instruction Multiple Thread (SIMT) fashion. It keeps many ALU (arithmetic logic unit) cores busy by hiding memory latency through zero-overhead thread switching. At any given clock cycle, multiple groups of threads (i.e. multiple of 32 or 64 threads) run in a Single Instruction Multiple Data (SIMD) fashion. When well optimized, data- and compute-intensive applications can be easily accelerated by several orders of magnitude. GPGPU is programmed using OpenCL [17] or CUDA [18] languages in which data- and compute-intensive portions of program are offloaded onto GPU device. The offloaded function-like code to be executed on the GPU device is called *kernel*. Host and device kernel codes can be executed either asynchronously or synchronously.

3.1. Limitations of Current GPGPU Paradigm

Although numerous applications have been successfully accelerated using GPUs with remarkable speedups, there are many other algorithms and applications that do not benefit from current GPGPU computing paradigm. This is because GPU hardware and software (i.e. programming model) are very different from conventional parallel platforms as they are evolved by the demand for real-time 3D graphics rendering. We summarize the big limitations of today's GPGPU computing that we also found present in our target application, DynEarthSol3D:

- **Data transfer overhead**: In conventional GPGPU settings, discrete GPU is physically connected through PCI-Express and has separate physical memory (see Figure 2).

Data to be used by kernel program must be copied to device memory before kernel execution. If an application consists of multiple sections of CPU and GPU computations that are interleaved and data-dependent on each other, frequent data transfer between host and device is necessary. Therefore, overall application performance is limited by the overhead associated with the slow data transfer.

- **Kernel launch overhead**: The host CPU communicates with GPU through device driver calls and each command including kernel invocation involves overhead. When a large number of kernel calls are performed throughout the program execution, the kernel launch overhead associated with device driver calls can be added up to significant portion of overall performance. Such overhead can be a serious problem especially when the kernel execution time is small. Therefore, launching multiple small kernels should be avoided whenever possible to reduce the overhead.

- **Irregular memory access**: GPU hardware architecture is designed for maximizing throughput for a group of threads rather than minimizing the latency of an individual thread. It implies that memory subsystem becomes very inefficient when threads issue memory requests with irregular access patterns [15]. Altering memory access patterns toward more hardware friendly ones is the most important yet challenging optimization to improve kernel performance. This is typically done by transforming data layout or changing computations in source code.

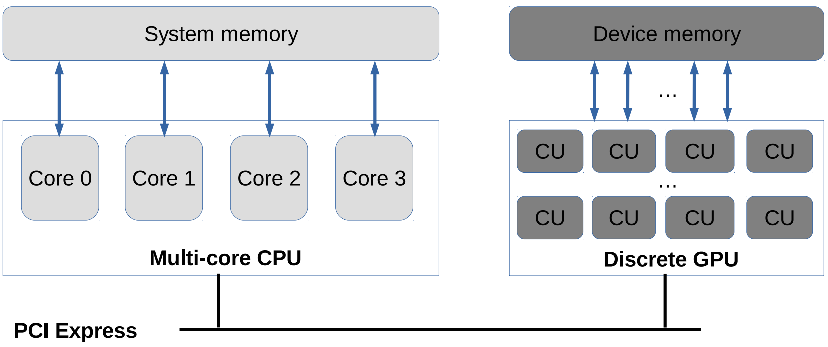


Figure 2: The system diagram of conventional discrete GPGPU platform.

3.2. Tightly Coupled CPU-GPU Heterogeneous Processors

Recent trend in microprocessor industry is to merge CPU and GPU on a single die [2, 21]. This is a natural choice in microprocessor industry as it offers numerous benefits such as fast interconnection and fabrication cost reduction. Recently, the processor industry and academic research community have formed a non-profit consortium, called HSA (Heterogeneous System Architecture) foundation [11] to define the standards of hardware and software for next generations of heterogeneous computing. Such processors that couple CPU and GPU at last level cache overcome some limitations of current GPGPU. Tightly coupled heterogeneous processors provide the following benefits.

- **Fast and fine-grained data sharing between CPU and GPU**: Multi-core CPU and many-core GPU are tightly coupled at last cache level on a single die and share a single physical memory (see Figure 3). This architecture design eliminates CPU-GPU data transfer overhead by sharing the same data.

- **Large memory support for GPU acceleration**: Data oriented applications such as big data processing and compute-intensive scientific simulations require a large memory space to minimize inefficient data copy back and forth. In tightly coupled heterogeneous processors, GPU device shares system memory that is typically a lot larger (e.g. 32GB) than device memory in discrete GPUs (e.g. 4GB).

- **Cache coherence between CPU and GPU**: This new hardware feature will remove off-chip memory access traffic significantly and allow fast, fine-grained data sharing between CPU and GPU. Both devices are capable of addressing the same coherent block of memory, supporting popular application design patterns such as producer-consumer [23].

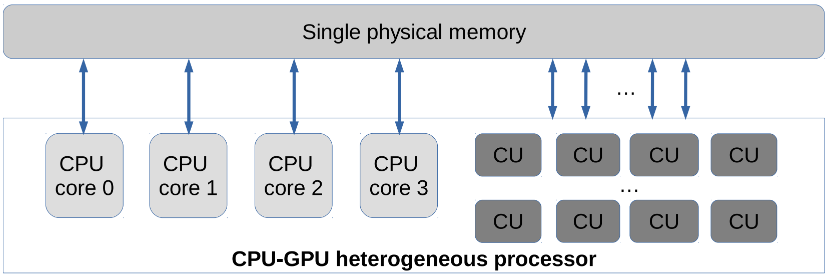


Figure 3: The system diagram of tightly coupled CPU-GPU heterogeneous processors.

4. Implementation and Optimization

As shown in Figure 1, DynEarthSol3D sequentially computes and updates unknowns on the nodes of a mesh in each time step. To identify time-consuming parts of the program, we first profile and decompose the execution time of serial version at functional level[2], and illustrate the result in Figure 4. This profiling and breakdown of execution time provides useful information that helps identify the candidate functions to be offloaded to GPU. Based on this information and through source code analysis, we have chosen following functions (or operations) to accelerate on GPU. They account for 88% of the total execution time in DynEarthSol3D. For the sake of readability of the paper, we use short names shown in paranthesis in Figure 4 in the rest of the paper.

[2]In serial DynEarthSol3D, computing and updating each property of mesh is written as a function. Note also that the remeshing operation is excluded from our analysis as it uses the external *Tetgen* library whose performance is not our focus in this work.

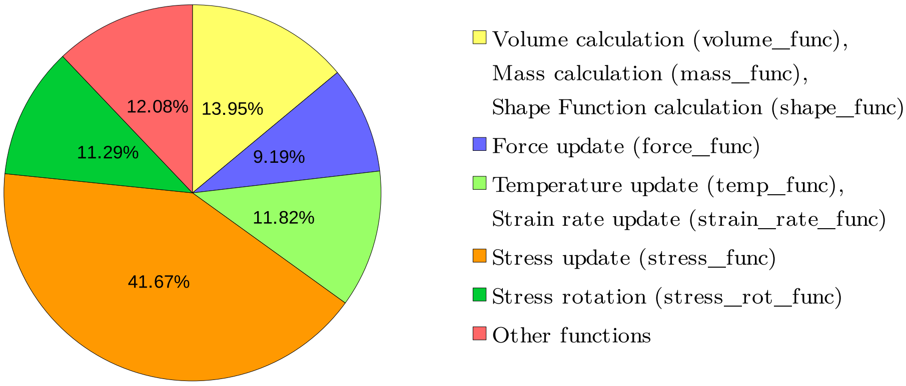


Figure 4: Performance breakdown of the serial version of DynEarthSol3D.

In the following subsections, we describe the general structure of our OpenCL implementation, followed by the detailed presentation of each optimization. Our optimizations focus on 1) memory access patterns, 2) data transfer between CPU and GPU, and 3) kernel launch overhead.

4.1. The Structure of OpenCL Implementation

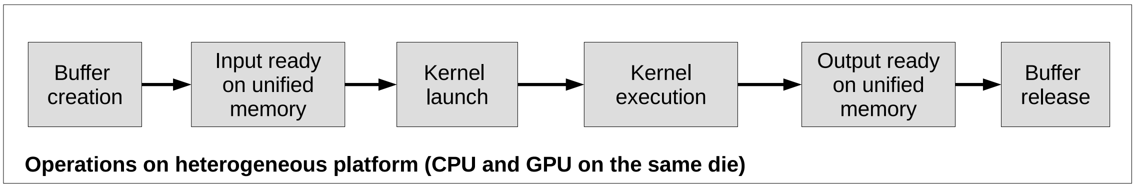


Figure 5: The structure of OpenCL implementation on CPU-GPU heterogeneous platform.

Initial GPU setup including device configuration, platform creation, and kernel building, etc. is performed only once before the program starts updating solutions in multiple time steps. The framework described in Figure 5 applies to all of our target functions. OpenCL buffers are first created with appropriate flags that enable the *zero copy* feature on heterogeneous processor. These buffers reside on the unified memory that is accessible to both CPU and GPU (This feature is detailed in Section 4.3). After buffers are created, kernel arguments are set up, and then kernel is launched. Each mesh element is processed by a specific thread through one-to-one mapping between work-item ID (in a n-dimensional thread index space called *NDRange*) and element ID. Multiple work-items in the NDRange are grouped into a work-group that is assigned to a compute unit on GPU. While each thread reads input element from global memory, processes, and updates it, two different elements mapped to two threads may share some nodes, which leads to a race condition when they update the shared node at the same time. To guarantee that outputs are updated correctly, the race condition must be handled through atomic operations. The execution on GPU continues until all threads complete their works and are synchronized by the host to ensure that output data are complete and valid. Finally, buffers are released, and the program continues its remaining operations in the current time step or moves on to the next one.

4.2. Memory Access Pattern Improvement

The performance of GPGPU is heavily dependent on the memory access behavior. This sensitivity is due to a combination of the underlying massively parallel processing execution model present on GPUs and the lack of architectural support to handle irregular memory access patterns. Hardware unfriendly memory accesses degrade performance significantly as it results the serialization of many expensive off-chip memory accesses. For linear and regular memory access patterns issued by a group of threads, the hardware coalesces them into a single (or fewer number of) memory transactions, which significantly reduce overall memory latency, consequently less thread stalls. Therefore, application performance can be significantly improved by minimizing irregularity of global memory access patterns.

In DynEarthSol3D, *Tetgen* program generates a mesh with a system of element and node numbers (IDs). Each tetrahedral element with its own ID is associated with four different nodes numbered in semi-random fashion. In our implementation, nodes are accessed sequentially by each thread. Therefore, the randomness of node IDs in an element results in irregular pattern of global memory accesses requested by a single thread which has to load and update node-related data locating randomly in global memory. Figure 6 illustrates a case where two adjacent elements may share three nodes (i.e. IDs 10, 30, 60) together. Figure 7a visualizes the randomness of the node system by representing each node ID corresponding to its element ID.

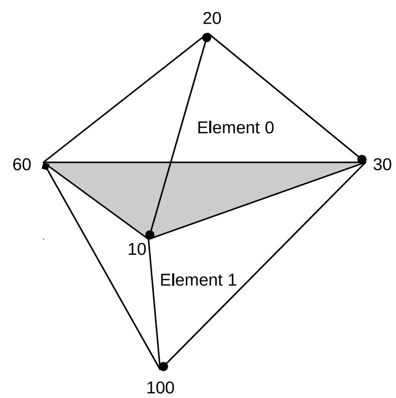
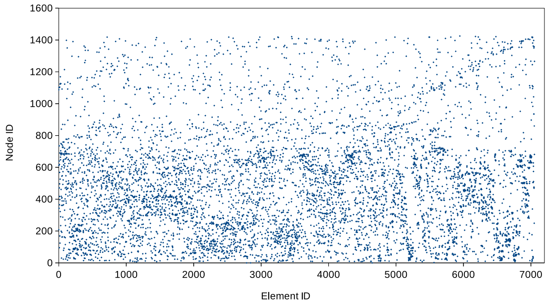
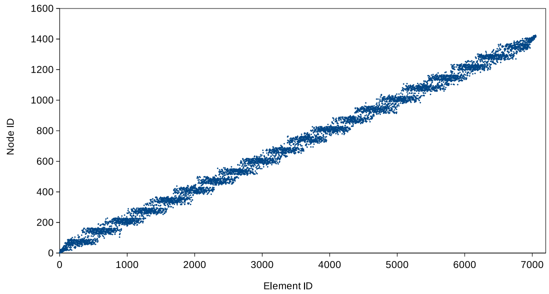


Figure 6: Two tetrahedral elements share three nodes.



(a) Original random relationship



(b) Improved relationship

Figure 7: Relationship between node and element IDs. Each element ID is mapped to a single thread in GPU kernel.

To eliminate the randomness of node ID system, we renumber all nodes so that they are ordered by their corresponding x coordinates and renumber all elements similarly by the x coordinates of their centers. As a result, node IDs within a single element and among multiple adjacent elements are close together. Figure 7b illustrates the improved relationship between node and element IDs. This improved pattern has a direct impact on memory access patterns of the kernel. Cache hit rate significantly increases, and memory accesses are coalesced. Therefore, overall memory latency during kernel execution is significantly decreased.

4.3. Data Transfer Elimination

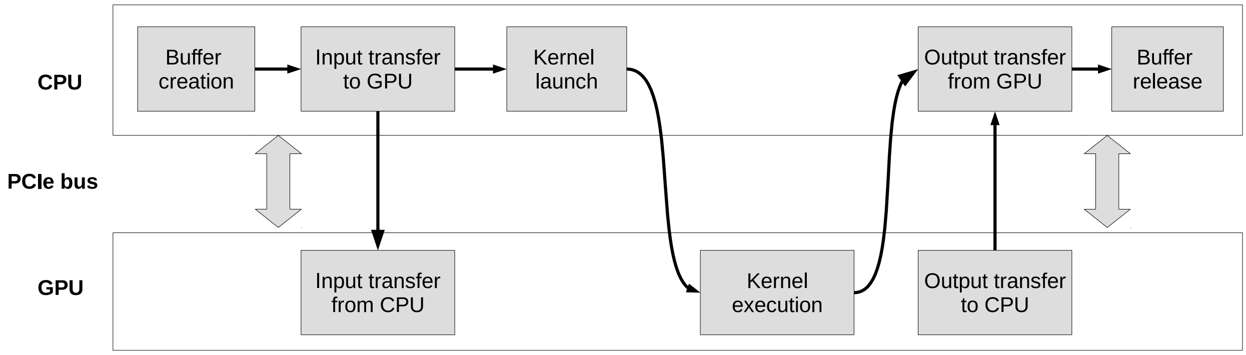


Figure 8: The structure of OpenCL implementation on discrete GPU platform.

Figure 8 shows the computational flow of OpenCL implementations on conventional discrete GPU platform with respect to physical execution hardware. On discrete GPU systems where CPU and GPU have separate memory subsystem, data copy between host and device must be done via low speed PCI Express bus. Such data movement takes considerable amount of time and can significantly offsets performance gains obtained by GPU kernel acceleration. This data copy is a serious problem in DynEarthSol3D as large amount of data has to be copied back and forth between host and device in each time step. Tightly coupled CPU-GPU heterogeneous processors offer a solution to this bottleneck as CPU and GPU share the same unified physical memory. Using a feature known as *zero copy*, data (or buffer) can be accessed by two processors without copying. Zero copy is enabled by passing one of following flags appropriately to *clCreateBuffer* OpenCL API function [1].

- **CL_MEM_ALLOC_HOST_PTR** Buffers created with this flag is "host-resident zero copy memory object" that is directly accessed by host at host memory bandwidth and visible to GPU device.
- **CL_MEM_USE_HOST_PTR** This flag is similar to CL_MEM_ALLOC_HOST_PTR. However, instead of allocating a new memory space belonging to either host or device, it uses a memory buffer that has been already allocated and currently being used by host program.
- **CL_MEM_USE_PERSISTENT_MEM_AMD** This flag is available only on AMD platform. It tells host program to allocate a new "device-resident zero copy memory object" that is directly accessed by GPU device and accessible in host code.

Because the first and third options allocate new empty memory spaces, the buffers need to be filled with data before kernel execution on GPU. In our DynEarthSol3D implementation, the second option is used to avoid such extra buffer setup. Figure 9 illustrates how both host program running on CPU and kernel running on GPU access data in shared buffers created with CL_MEM_USE_HOST_PTR flag.

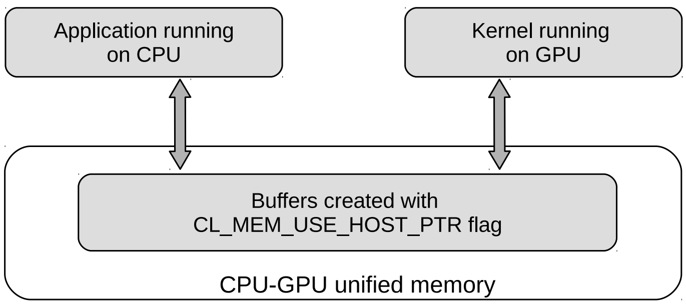


Figure 9: Memory buffer shared by CPU and GPU (a.k.a. zero copy) on tightly coupled heterogeneous processors.

4.4. Kernel Launch Overhead Minimization

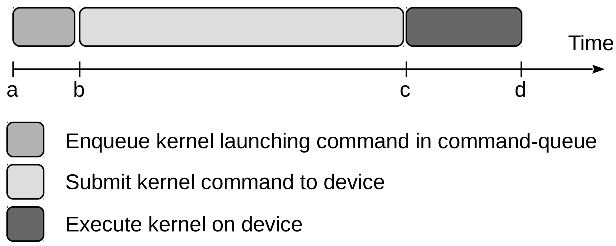


Figure 10: The breakdown of kernel execution process.

Executing a kernel on GPU device consists of three steps. A kernel launch command is first enqueued to the command-queue associated with device, and then the command is submitted to device before the kernel is finally executed. Queuing and submitting kernel launch command are considered as overhead. According to our experiment on AMD platforms, the command submission (second block in Figure 10) accounts for most of the overhead. This overhead becomes significant when actual kernel execution time is relatively short compared to kernel queuing and submission time as exemplified in Figure 10. In addition, the use of CL_MEM_USE_HOST_PTR flag results in a small runtime overhead as the size of buffers used in kernel increases. These two kinds of overhead are repeatedly accumulated in DynEarthSol3D as kernels are re-launched in each time step.

The only available solution to this overhead is to reduce the number of kernel launches throughout the program execution. Toward that end, we merge multiple functions into a single kernel, so there are less number of kernel to be launched. If data dependency exists between two functions (meaning that the second function needs to use outputs of the first one), they cannot be merged into a single kernel because GPGPU programming and hardware execution model does not guarantee that the first function finishes its entire thread execution before the second starts. Without such guarantee, the second function might use old input data that has not been updated yet by the first one.

In our implementation, we first perform in-depth data dependency analysis of DynEarthSol3D to identify possible combinations of functions with no data dependency. Based on this analysis, we find two combinations. The first one combines *volume_func*, *mass_func* and *shape_func*. The other merges *temp_func* and *strain_rate_func*. For simplicity, we call the first and second merged kernels *intg_kernel_1* and *intg_kernel_2* respectively in the rest of this paper.

5. Experimental Results

To evaluate the performance of our proposed OpenCL implementation on tightly coupled CPU-GPU heterogeneous processors we compare its performance with both serial and OpenMP-based implementations as baselines. We also analyze the impact of each proposed optimization technique.

In all experiments, we use the same program configuration with varying sizes of mesh of elasto-plastic material. The program runs in 1,000 time steps, and its outputs are written into output files every 100 steps. Performance results are accumulated in each time step. Wall clock timer and OpenCL profiler are used to measure performance of host code and kernel respectively. We compare our OpenCL output results with serial version's outputs to verify the correctness of our implementation.

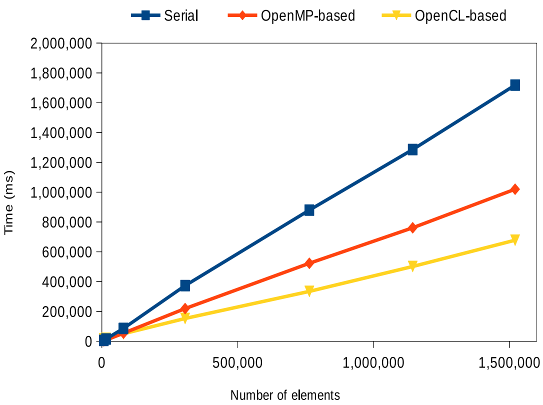
We experiment our OpenCL-based implementation on AMD APU A10-7850K which is the latest heterogeneous processor as of this paper writing. This tightly coupled heterogeneous processor consists of a quad-core CPU with maximum clock speed of 4.0 GHz and a Radeon R7 GPU with eight compute units running at 720 MHz on the same die. Our baseline versions (i.e., serial and OpenMP-based implementation) are tested on the quad-core CPU of the same APU for fair comparison. In evaluating the impact of data transfer elimination, we use a high-end discrete AMD GPU Radeon HD 7970 codenamed Tahiti. It has 32 compute units with maximum clock speed of 925 MHz. The operating system is 64-bit Ubuntu Linux 12.04 and AMD APP SDK 2.9 (OpenCL 1.2) is used.

5.1. Overall Acceleration

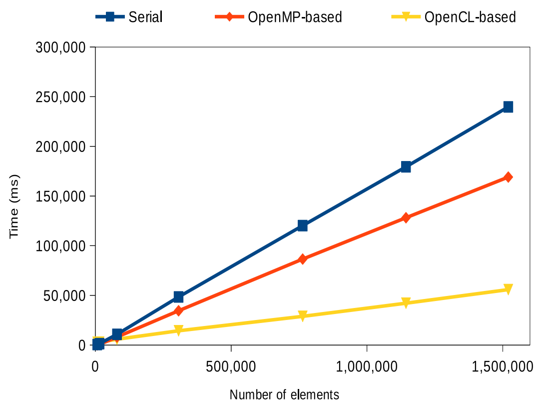
In this section, we compare the performance of our OpenCL-based implementation with two baseline versions: serial and OpenMP-based implementations at both program- and function-levels. The results[3] are shown in Figure 11. We varied the number of mesh elements from 7 thousand to 1.5 million. Regarding integrated kernels *intg_kernel_1* and *intg_kernel_2*, we do comparisons with their corresponding component functions in serial and OpenMP-based implementations. Note that *intg_kernel_1* merges *volume_func*, *mass_func* and *shape_func*, and *intg_kernel_1* merges *temp_func* and *strain_rate_func*.

At program level, our OpenCL implementation optimized for tightly coupled heterogeneous processor outperforms both serial and OpenMP-optimized versions by 154% and 50% respectively for 1.5-million-element mesh. At function level, all target functions show a similar trend in performance. Especially, integrated kernel *intg_kernel_1* is 329% and 203% faster than its before-merged case in serial and OpenMP versions respectively. The impact of merging kernels is analyzed in more detail in later section.

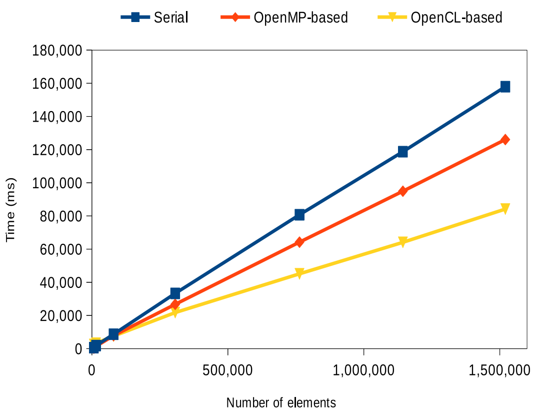
[3]For fair comparisons, experiments are done with the improved node ID system. A less random memory access pattern also improves the performance of serial and OpenMP-based versions due to better cache utilization on CPU.



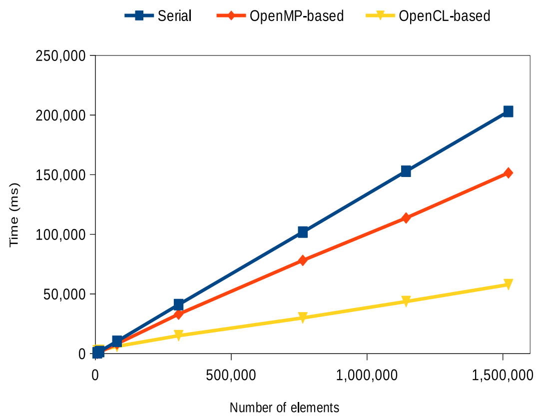
(a) Overall performance (at program level).



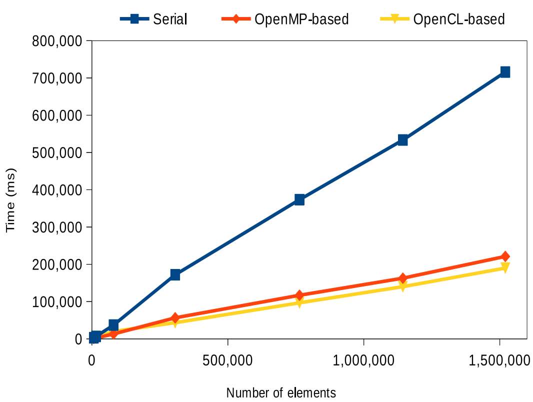
(b) Performance of *intg_kernel_1*.



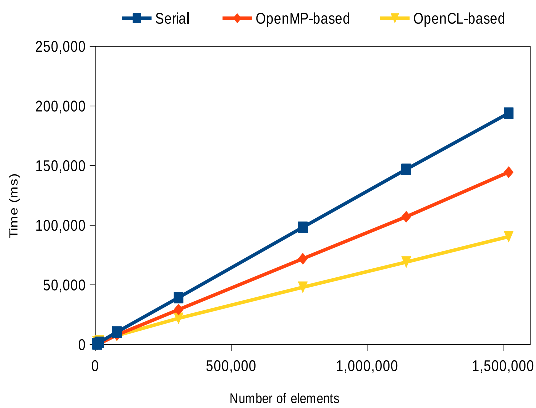
(c) Performance of *force_func*.



(d) Performance of *intg_kernel_2*.



(e) Performance of *stress_func*.



(f) Performance of *stress_rot_func*.

Figure 11: Performance comparison among different implementations.

An interesting observation from these comparisons is that performance gain from GPU acceleration becomes more substantial as input size increases. If there are a small number of threads issued (i.e. small input), GPU computing hardware resources are underutilized and unable to compensate for the setup overhead of GPU hardware pipelines.

5.2. Impact of Memory Access Pattern Optimization

In this section, we analyze the impact of node ID system improvement on performance by comparing two implementations with and without this improvement at function level. Only kernel execution time measured by AMD profiler is concerned here as memory access pattern does not affect other parts of our optimization. Figure 12 illustrates these performance comparisons.

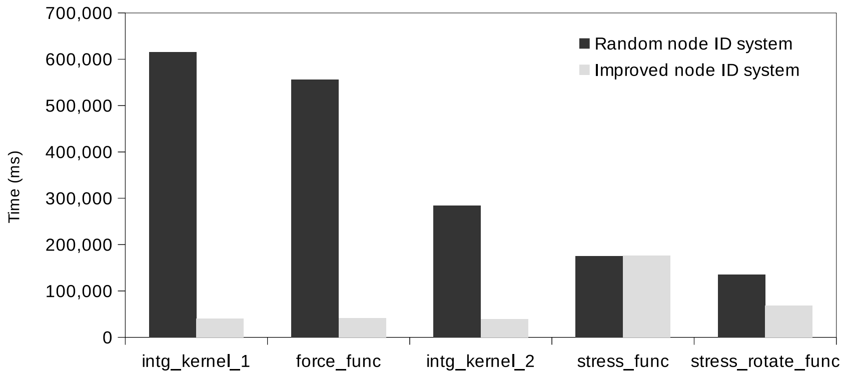


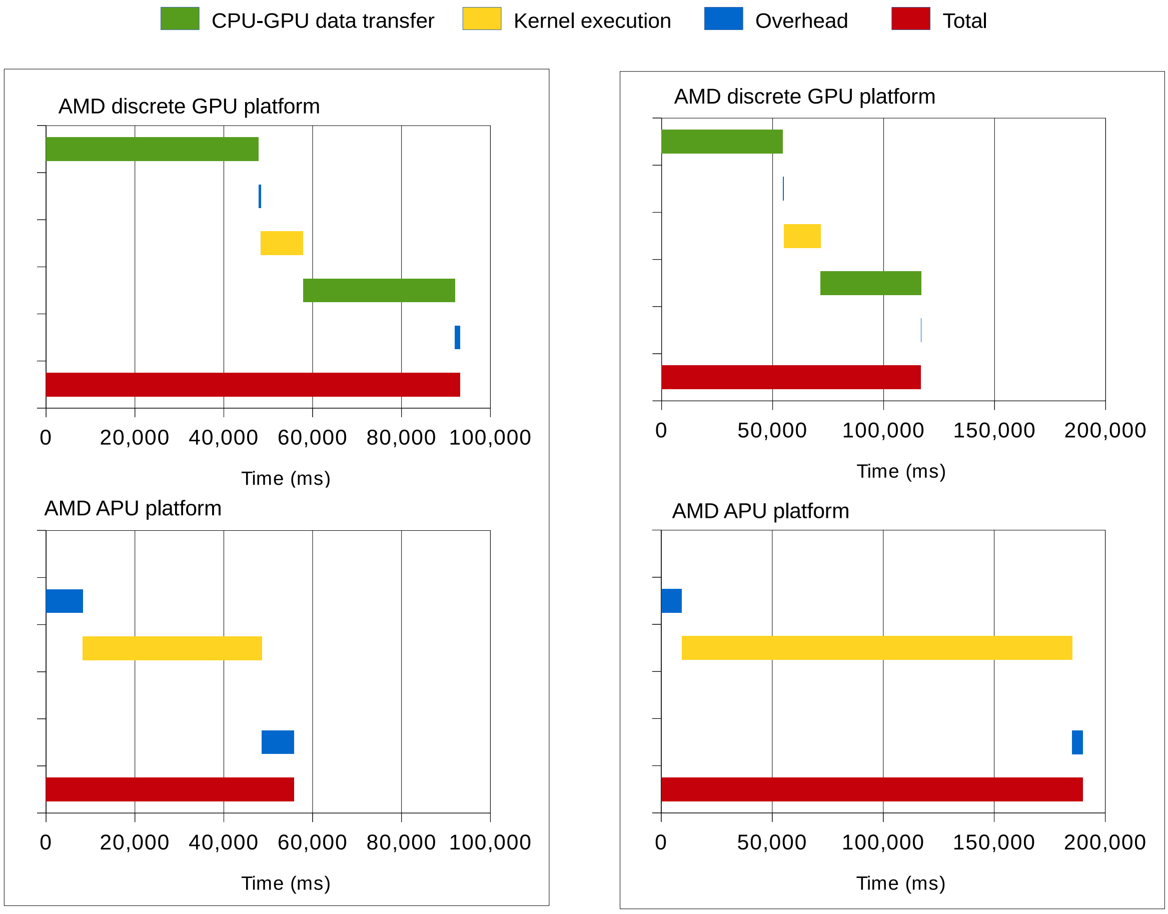
Figure 12: The impact of memory access pattern on kernel execution time.

Substantial improvement in kernel execution is achieved in functions that process node-related mesh properties intensively (i.e., 15x, 13.4x, 7.2x, 2x speedups in *intg_kernel_1*, *force_func*, *intg_kernel_2*, and *stress_rot_func* respectively). The randomness of node IDs does not affect performance of *stress_func* because it deals with only element-related mesh properties. The improved memory access patterns enable kernels to take advantage of spatial locality within a thread and across threads in a work-group, which consequently yields better utilization of GPU cache system. In addition, because multiple global memory requests can be coalesced into fewer number of memory transactions, the improved node ID pattern reduces both on-chip and off-chip memory traffic substantially and reduces overall memory latency.

5.3. Impact of Data Transfer Elimination

In order to demonstrate the significant benefit of utilizing tightly coupled CPU-GPU heterogeneous processors in terms of data transfer overhead, we present and compare the execution time of two kernels: *intg_kernel_1* and *stress_func* in Figure 13. Both functions shown here compute and process large number of mesh properties that are associated with a considerable amount of data. We test them on 1) high-end discrete GPU Radeon HD

7970 (codenamed Tahiti) with explicit data transfer by calling *clEnqueueWriteBuffer* and *clEnqueueReadBuffer*, and 2) heterogeneous processor AMD APU A10-7850K with *zero copy* feature enabled with respect to data transfer, kernel execution and overhead.



(a) *intg_kernel_1* function.

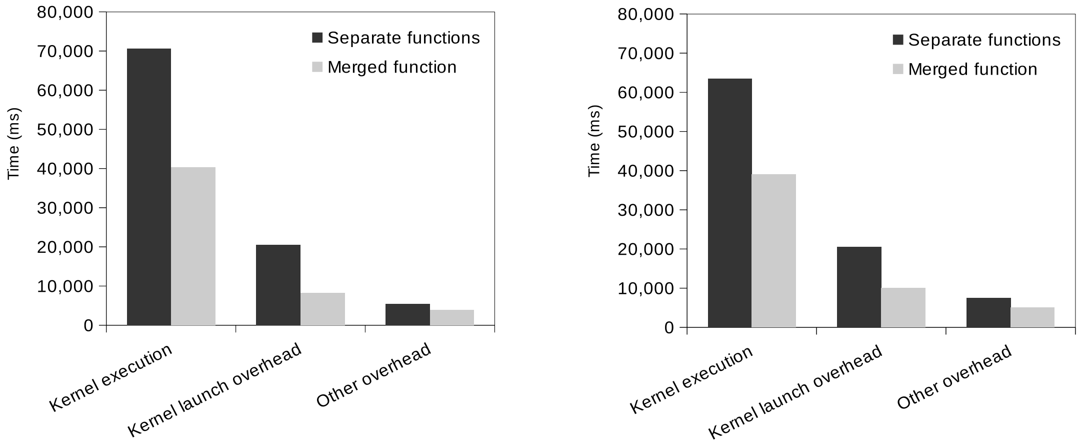
(b) *stress_func* function.

Figure 13: The impact of data transfer elimination.

On discrete GPU, explicit data transfer between host and device memory accounts for 88% and 85% of total performance in *intg_kernel_1* and *stress_func* respectively. The reason for this extremely high cost of data communication on discrete platform is that all data are transferred through PCI Express bus at slow speed. In contrast, there is no data transfer on CPU-GPU heterogeneous platform. Regarding *intg_kernel_1* function, the AMD APU outperforms the high-end discrete GPU (i.e. 67% faster) despite the fact that the Tahiti GPU is provided with more powerful computing capability (more compute units and higher clock speed than the AMD APU). However, in the case of *stress_func* function, the elimination of data transfer is not enough to compensate for the much less computing capability of AMD APU. The reason is that compared to *intg_kernel_1* function, *stress_func*'s kernel executes a lot more arithmetic computations that the discrete GPU is capable of performing much faster than the embedded GPU of heterogeneous processor is able to do.

5.4. Impact of Kernel Overhead Minimization

According to our experiment, the kernel launch overhead accounts for 27% of *intg_kernel_1*'s total execution time on the CPU-GPU heterogeneous processor. This section demonstrates the benefits of our overhead minimization technique in two integrated kernels: *intg_kernel_1* and *intg_kernel_2*. By comparing them with their separate versions, we notice that performance gain comes from two aspects: reduced overhead and improved total kernel execution.



(a) *intg_kernel_1* function. (b) *intg_kernel_2* function.

Figure 14: The impact of kernel overhead minimization.

Figure 14 shows the performance comparison between the two merged functions and their corresponding separate versions. The results show that the overhead is reduced by 53% and 46% in both *intg_kernel_1* and *intg_kernel_2* respectively by merging kernels into a single kernel. Moreover, merging kernels also speeds up the kernel execution (1.8x and 1.6x respectively). By merging kernels, data can be reused across different individual kernels, which reduces global memory accesses. Moreover, merging multiple kernels into a single kernel increases the number of arithmetic computations that better hide memory latency [1].

6. Conclusions and Future Works

In this paper, we present the acceleration of DynEarthSol3D on tightly coupled CPU-GPU heterogeneous processors by leveraging their new features, and compare its performance and benefits with other serial and parallel alternatives. Our results show that the OpenCL-based implementation on tightly coupled heterogeneous processors outperforms both serial and OpenMP-based implementations that run on a multi-core CPU. We also emphasize the importance of memory access pattern in GPGPU programming. With a proper node ID system that reduces the randomness of global memory accesses, memory latency is decreased significantly in our OpenCL-based optimization. Furthermore, *zero copy* feature that is available on heterogeneous platform solves the big issue of expensive data transfer between host and device memory in conventional discrete GPU. Such benefits are examined in our in-depth analysis. We also discuss how integrating multiple small functions into a single kernel reduces both overhead and kernel execution time.

Our work demonstrates the potential of tightly coupled CPU-GPU heterogeneous processors for the acceleration of data- and compute-intensive programs such as DynEarthSol3D. However, some issues of current heterogeneous processors need to be addressed in the future. The computing power of embedded GPU in current heterogeneous processors (e.g. 8 compute units in Kaveri) is much lower than the one of discrete GPUs (e.g. 32 compute units in Tahiti). This gap imposes a trade-off between better kernel performance on discrete platforms and "zero" data transfer on heterogeneous processors. In the future, heterogeneous processors are expected to provide more powerful compute units. Moreover, although the need for data transfer is eliminated, high overhead observed on AMD's heterogeneous platform in our experiment needs to be eliminated. This problem can be addressed with better software supports (i.e. driver) from hardware vendor. Currently, OpenCL 1.2 does not support all promising HSA features of heterogeneous computing. With the OpenCL 2.0 coming soon, we are looking forward to utilizing these new features in our future optimization of DynEarthSol3D.

7. References

[1] AMD, 2013. AMD Accelerated parallel processing: OpenCL programming guide.
URL http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf

[2] AMD, 2014. AMD A-Series APU Processors.
URL http://www.amd.com/en-gb/products/processors/desktop/a-series-apu

[3] AMD, 2014. AMD Accelerated Processing Units (APUs).
URL http://www.amd.com/en-us/innovations/software-technologies/apu

[4] Behn, M. D., Ito, G., Aug. 2008. Magmatic and tectonic extension at mid-ocean ridges: 1. Controls on fault characteristics. Geochemistry Geophysics Geosystems 9 (8), Q08O10.
URL http://www.agu.org/pubs/crossref/2008/2008GC001965.shtml

[5] Braun, J., Thieulot, C., Fullsack, P., DeKool, M., Beaumont, C., Huismans, R., 2008. DOUAR: A new three-dimensional creeping flow numerical model for the solution of geological problems. Phys. Earth Planet. Int. 171 (1-4), 76–91.
URL http://www.sciencedirect.com/science/article/B6V6S-4SJ2WV0-3/2/94d7290704141e50e939397d3c352cd6

[6] Buck, W. R., Lavier, L. L., Poliakov, A. N. B., Apr. 2005. Modes of faulting at mid-ocean ridges. Nature 434 (7034), 719–23.
URL http://www.nature.com/nature/journal/v434/n7034/full/nature03358.html

[7] Choi, E., Gurnis, M., 2008. Thermally induced brittle deformation in oceanic lithosphere and the spacing of fracture zones. Earth Planet. Sci. Lett. 269, 259–270.

[8] Choi, E., Tan, E., Lavier, L. L., Calo, V. M., May 2013. DynEarthSol2D: An efficient unstructured finite element method to study long-term tectonic deformation. Journal of Geophysical Research: Solid Earth 118 (5), 2429–2444.
URL http://doi.wiley.com/10.1002/jgrb.50148

[9] Čuma, M., Zhdanov, M. S., 2014. Massively parallel regularized 3d inversion of potential fields on cpus and gpus. Computers & Geosciences 62, 80–87.

[10] Cundall, P. A., Board, M., 1988. A microcomputer program for modeling large strain plasticity problems. In: Swoboda, G. (Ed.), Numerical Methods in Geomechanics:. pp. 2101–2108.
[11] HSA Foundation, 2013. Heterogeneous System Architecture (HSA) Foundation.
URL http://hsafoundation.com/
[12] Huet, B., Le Pourhiet, L., Labrousse, L., Burov, E., Jolivet, L., Feb. 2011. Post-orogenic extension and metamorphic core complexes in a heterogeneous crust: the role of crustal layering inherited from collision. Application to the Cyclades (Aegean domain). Geophysical Journal International 184 (2), 611–625.
URL http://doi.wiley.com/10.1111/j.1365-246X.2010.04849.x
[13] Intel, 2014. Intel Core Processor Family.
URL http://www.intel.com/content/www/us/en/processors/core/core-processor-family.html
[14] Ito, G., Behn, M. D., Sep. 2008. Magmatic and tectonic extension at mid-ocean ridges: 2. Origin of axial morphology. Geochemistry Geophysics Geosystems 9 (9).
URL http://www.agu.org/pubs/crossref/2008/2008GC001970.shtml
[15] Jang, B., Schaa, D., Mistry, P., Kaeli, D., 2011. Exploiting memory access patterns to improve memory performance in data-parallel architectures. Parallel and Distributed Systems, IEEE Transactions on 22 (1), 105–118.
[16] Lyakhovsky, V., Segev, A., Schattner, U., Weinberger, R., Jan. 2012. Deformation and seismicity associated with continental rift zones propagating toward continental margins. Geochemistry Geophysics Geosystems 13.
URL http://www.agu.org/pubs/crossref/2012/2011GC003927.shtml
[17] Munshi, A., et al., 2009. The opencl specification. Khronos OpenCL Working Group 1, l1–15.
[18] Nvidia, C., 2014. Programming guide.
URL http://docs.nvidia.com/cuda/cuda-c-programming-guide
[19] Poliakov, A., Buck, W. R., 1998. Mechanics of stretching elastic-plastic-viscous layers: Applications to slow-spreading mid-ocean ridges. In: Buck, W. R., Delaney, P. T., Karson, J. A., Lagabrielle, Y. (Eds.), Faulting and Magmatism at Mid-Ocean Ridges. Vol. 106 of AGU Monograph. AGU, Washington D.C., pp. 305–324.
[20] Poliakov, A. N. B., Cundall, P. A., Podladchikov, Y. Y., Lyakhovsky, V. A., 1993. An explicit inertial method for the simulation of viscoelastic flow: An evaluation of elastic effects on diapiric flow in two- and three-layers models. In: Stone, D. B., Runcorn, S. K. (Eds.), Flow and Creep in the Solar Systems: Observations, Modeling and Theory. Kluwer Academic Publishers, pp. 175–195.
[21] Shevtsov, M., 2013. OpenCL: Advantages of the Heterogeneous Approach - Intel Developer Zone.
URL https://software.intel.com/en-us/articles/opencl-the-advantages-of-heterogeneous-approach
[22] Si, H., TetGen, A., 2006. A quality tetrahedral mesh generator and three-dimensional delaunay triangulator. Weierstrass Institute for Applied Analysis and Stochastic, Berlin, Germany.
[23] Su, L. T., 2013. Architecting the future through heterogeneous computing. In: Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2013 IEEE International. IEEE, pp. 8–11.
[24] Tahmasebi, P., Sahimi, M., Mariethoz, G., Hezarkhani, A., 2012. Accelerating geostatistical simulations using graphics processing units (gpu). Computers & Geosciences 46 (0), 51 – 59.
URL http://www.sciencedirect.com/science/article/pii/S0098300412001240
[25] Tan, E., Choi, E., Lavier, L., Calo, V., 2013. DynEarthSol3D: An Efficient and Flexible Unstructured Finite Element Method to Study Long-Term Tectonic Deformation,. Abstract DI31A-2197 presented at 2013 Fall Meeting, AGU, San Francisco, Calif., 9-13 Dec.