Accelerating DynEarthSol3D on tightly coupled CPU–GPU heterogeneous processors

Tuan Ta\textsuperscript{a,},*, Kyoshin Choo\textsuperscript{a}, Eh Tan\textsuperscript{b}, Byunghyun Jang\textsuperscript{a,}\textsuperscript{,}, Eunseo Choi\textsuperscript{c}

\textsuperscript{a} Department of Computer and Information Science, The University of Mississippi, MS, USA
\textsuperscript{b} Institute of Earth Sciences, Academia Sinica, Taipei, Taiwan
\textsuperscript{c} Center for Earthquake Research and Information, The University of Memphis, Memphis, TN, USA

\textbf{ARTICLE INFO}

Article history:
Received 13 October 2014
Received in revised form 17 February 2015
Accepted 5 March 2015
Available online 6 March 2015

Keywords:
Computational Tectonic Modeling
Long-term lithospheric deformation
Heterogeneous computing
GPGPU
Parallel computing

\textbf{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 small input mesh containing around 3000 elements in 20 million time steps would take more than 10 days on a 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 in the heterogeneous processor has significantly less number of cores than high-end discrete GPU.

\textcopyright 2015 Elsevier Ltd. All rights reserved.

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 codes following the Fast Lagrangian Analysis of Continua (FLAC) algorithm (Cundall and Board, 1988). 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., Behn and Ito, 2008; Buck et al., 2005; Choi and Gurnis, 2008; Huet et al., 2011; Ito and Behn, 2008; Lyakhovsky et al., 2012; Poliakov and Buck, 1998; Poliakov et al., 1993). 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. (2013) created a new code, DynEarthSol2D, and Tan et al. (2013) further extended it to three dimensions, DynEarthSol3D. DynEarthSol3D (Dynamic Earth Solver in three (3) 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\textsuperscript{1} under the terms of the MIT/X Windows System

\textsuperscript{1} http://bitbucket.org/tan2/dynearthsol3d

\url{http://dx.doi.org/10.1016/j.cageo.2015.03.003}

0098-3004/© 2015 Elsevier Ltd. All rights reserved.
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. The processing speed is 0.5 μs per tetrahedra element per time step for optimized serial implementation on a high-end CPU. The running time would go up linearly with the number of elements of a mesh (typically 10^5–10^6 elements) and the number of time steps (typically 10^7–10^8 steps). This huge amount of running time limits both mesh size and resolution during development and in practice.

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 off-loading 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 memory and host system memory diminishes the performance gain obtained by GPU acceleration. Deteriorated by kernel launch time, such overheads can be a serious bottleneck of a program. 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 (AMD, 2014b; Intel, 2014). Such tightly coupled CPU–GPU heterogeneous processors provide solutions to several limitations of traditional discrete GPUs such as aforementioned data transfer overhead (Cuma and Zhdanov, 2014; Tahmasebi et al., 2012), limited GPU device memory (i.e., GDDR) size (Tahmasebi et al., 2012), and disjoint memory address space. CPU and GPU on tightly coupled heterogeneous processors share the same data in unified physical memory (data transfer is no longer needed) which is typically a lot larger (e.g., 32 GB) than discrete GPU device memory (e.g., 4 GB).

In this paper, we present the acceleration of DynEarthSol3D on tightly coupled CPU–GPU heterogeneous processors, leveraging 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. The result is encouraging because the computing power of embedded GPU on the heterogeneous processor we tested is less than one fourth of that of discrete GPU 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 GPU 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

Fig. 1 visualizes the computational flow of DynEarthSol3D. First, a mesh composed of tetrahedral elements is created by an external mesh generator named Tetgen (Si and TetGen, 2006). Each element of the mesh consists of four nodes which function as interpolation points for unknown variables such as velocity and temperature. With a mesh resolution of 1 km, the program needs around 50 time steps to perform 1 year of evolution. The time step size is linearly proportional to the mesh resolution. The program runs through many 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 velocity field, 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 net 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. Subsequently, element volume, mass and shape functions are updated. At this stage, the program checks if accumulated deformation has distorted the mesh too severely. If so, Tetgen is invoked again 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 (Braun et al., 2008). After the new mesh is created and variables are mapped onto the new mesh, element volume, mass and shape functions are 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 using 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 \( 0, 2, \ldots, 2(P-1) \) are first processed in parallel, with each set covered by a thread. Then, after all threads finish processing, elements in set \( 1, 3, \ldots, 2P-1 \) 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 (Munshi et al., 2009) or CUDA (Nvidia, 2014) 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 major 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 Fig. 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 slow data transfer.
- **Kernel launch overhead**: 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 (Jang et al., 2011). 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.

3.2. Tightly coupled CPU–GPU heterogeneous processors

Recent trend in microprocessor industry is to merge CPU and GPU on a single die (AMD, 2014a; Shevtsov, 2013). This is a natural choice in microprocessor design 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 Heterogeneous System Architecture (HSA Foundation, 2013) 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 Fig. 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., 32 GB) than device memory in discrete GPUs (e.g., 4 GB).
- **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 (Su, 2013).

4. Implementation and optimization

As shown in Fig. 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 running in 20 million time steps at functional level with a specific input mesh size (e.g., around 3000 tetrahedral elements). The profiling information is illustrated in Fig. 4. The profiling and breakdown of execution time provide useful information that helps identify the candidate functions to be offloaded to GPU. Based on this

---

2 Remeshing function accounts for less than 4% of total running time in our experiment with a typical remeshing interval of 100,000 time steps. As its contribution to overall running time in typical cases is small and it uses the external Tetgen library whose performance is not our focus in this work, the remeshing operation is excluded from our analysis.
information and through source code analysis, we have chosen following functions (or operations) to accelerate on GPU. They account for 88.42% of the total execution time in DynEarthSol3D. For the sake of readability of the paper, we use short names shown in parenthesis in Fig. 4 in the rest of the paper.

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

Initial GPU setup including device configuration, platform creation, and kernel building is performed only once before the program starts updating solutions in multiple time steps. The framework described in Fig. 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 by both CPU and GPU. (This feature is detailed in Section 4.3.) After buffers are created and kernel arguments are set up, 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 them 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. Fig. 6 illustrates a case where two adjacent elements may share three nodes (i.e., IDs 10, 30, 60) together. Fig. 7a visualizes the randomness of the node system by representing each node ID corresponding to its element ID.

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. Fig. 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

Fig. 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
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 numbers of tetrahedral elements in the input mesh of elasto-plastic material. The program runs in 1000 time steps, illustrating 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.
function-levels. The results\textsuperscript{3} are shown in Figs. 11 and 12. We varied the number of mesh elements from 7000 to 1.5 million. Regarding integrated kernels \texttt{intg\_kernel\_1} and \texttt{intg\_kernel\_2}, we do comparisons with their corresponding component functions in serial and OpenMP-based implementations. Note that \texttt{intg\_kernel\_1} merges \texttt{volume\_func}, \texttt{mass\_func} and \texttt{shape\_func}, and \texttt{intg\_kernel\_1} merges \texttt{temp\_func} and \texttt{strain\_rate\_func}.

At application 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 \texttt{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.

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 the application. Fig. 13 illustrates these performance comparisons.

Substantial improvement in kernel execution is achieved in functions that process node-related mesh properties intensively (i.e., 15 $\times$ 13.4 $\times$ 72 $\times$ 2 $\times$ speedups in \texttt{intg\_kernel\_1}, \texttt{force\_func}, \texttt{intg\_kernel\_2}, and \texttt{stress\_rot\_func} respectively). The randomness of node IDs does not affect performance of \texttt{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: \texttt{intg\_kernel\_1} and \texttt{stress\_func} in Fig. 14. Both

\textsuperscript{3} This is total execution time from the beginning to the end of the application excluding only remeshing and initialization which are not significant (i.e., less than 4\% and much less than 0.01\% respectively in the serial version).

\textsuperscript{4} For fair comparisons, experiments are conducted 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.
Fig. 11. Performance comparison among different implementations: (a) Overall performance (at application 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.

Fig. 12. Performance breakdown of OpenCL-optimized implementation (i.e., with a 1.5-million-element input mesh) of DynEarthSol3D.

- Volume calculation (volume_func)
- Mass calculation (mass_func)
- Shape Function calculation (shape_func)
- Force update (force_func)
- Temperature update (temp_func)
- Strain rate update (strain_rate_func)
- Stress update (stress_func)
- Stress rotation (stress_rot_func)
- Other functions (remeshing is excluded)
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 A10-7850 K with zero copy feature properly implemented with respect to data transfer, kernel execution and overhead.

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 slow PCI Express bus. 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 by 67% despite the fact that the Tahiti GPU has 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` kernel executes a lot more arithmetic computations that the discrete GPU is capable of performing much faster than the embedded GPU of heterogeneous processor.

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 proposed 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.

Fig. 15 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 (1.8× and 1.6× 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 helps the hardware better hide memory latency (AMD, 2013).

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 quantified 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, current heterogeneous processors have some issues that 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 removed. We also note that upcoming official driver from AMD would improve the overhead and support the AMD’s heterogeneous platform better. Currently, OpenCL 1.2 does not support all promising HSA features of heterogeneous computing. With the upcoming OpenCL 2.0, we expect to utilize these new features in our future optimization of DynEarthSol3D.

Acknowledgements

The authors would like to thank Dave May and anonymous reviewer for their valuable comments. This research is supported in part by National Science Foundation (NSF) grant CCF-1337138.

References


