Accelerating Kirchhoff Migration on GPU using Directives

Rengan Xu[†], Maxime Hugues[‡], Henri Calandra[‡], Sunita Chandrasekaran[†], Barbara Chapman[†]
[†]*Department of Computer Science, University of Houston, Houston, TX, USA*
Email:{rxu6,schandrasekaran,bchapman}@uh.edu
[‡]*TOTAL E&P Research and Technology USA, Houston, TX, USA*
Email:{maxime.hugues,henri.calandra}@total.com

***Abstract*—Accelerators offer the potential to significantly improve the performance of scientific applications when offloading compute intensive portions of programs to the accelerators. However, effectively tapping their full potential is difficult owing to the programmability challenges faced by the users when mapping computation algorithms to the massively parallel architectures such as GPUs.**

Directive-based programming models offer programmers an option to rapidly create prototype applications by annotating region of code for offloading with hints to the compiler. This is critical to improve the productivity in the production code. In this paper, we study the effectiveness of a high-level directive-based programming model, OpenACC, for parallelizing a seismic migration application called Kirchhoff Migration on GPU architecture. Kirchhoff Migration is a real-world production code in the Oil & Gas industry. Because of its compute intensive property, we focus on the computation part and explore different mechanisms to effectively harness GPU's computation capabilities and memory hierarchy. We also analyze different loop transformation techniques in different OpenACC compilers and compare their performance differences. Compared to one socket (10 CPU cores) on the experimental platform, one GPU achieved a maximum speedup of 20.54x and 6.72x for interpolation and extrapolation kernel functions.

***Keywords*-OpenACC; Kirchhoff Migration; GPU; Directives; Programming Model;**

I. Introduction

Heterogenous architectures comprising of CPU processors and computation accelerators such as GPUs have been widely used in the High Performance Computing (HPC) field. These architecutres not only preserve CPU's powerful control capability, but also provide massively parallel computing capabilities. A wide variety of applications have been ported to such architectures and reasonable speedup have been achieved.

Currently the two main stream low-level programming models for the GPU architectures are CUDA [2] and OpenCL [7]. These models offer users programming interfaces with execution models closely matching the GPU architectures. Although GPU programming is achieveable with these low-level models, the programmers are required to thoroughly understand the hardware and significantly change the application structure and algorithm. To improve the productivity of application porting, an alternate approach is to use high-level directive-based programming models such as OpenACC [6], OpenMP 4.0 [8] and HMPP [9]. These models can be used by inserting directives and runtime calls into the existing source code and rely on compilers to transform and optimize the code internally using lower-level programming models.

In this paper, we focus on OpenACC programming model, an open standard for accelerator programming, to explore the code portability and performance in a production quality seismic migration application called Kirchhoff Migration. Kirchhoff Migraiton [18] is a widely used application for subsurface imaging in the Oil & Gas industry. The computation intensive and inherent parallel properties make this application an ideal candidate for the GPU architecture. We discuss how we parallelize and tune the performance of this application using the OpenACC model, and present results with different loop transformation techniques from different compilers.

The main contributions of this paper include:

- We present the parallelization techniques that are specific to Kirchhoff Migration using directive-based approach.
- Based on the application's properties, we harness GPU's compute capabilities and memory hierarchy to optimize the performance.
- We investigate the generated kernel by a vendor compiler, and use the kernel transformation technique from an open source compiler, and compare the performance between these two different implementations.

The organization of this paper are as follows: Section II provides an overview of the Kirchhoff Migration and Section III gives an overview of GPU architecture and OpenACC directives. Section IV describes the parallelization details. The preliminary performance results are discussed in Section V. Section VI highlights the related work in this area. We conclude our work in Section VII.

II. Overview of Kirchhoff Migration

Kirchhoff Migration is an algorithm to obtain 3D images of the earth subsurface. The working idea behind this algorithm is as follows. During a seismic survey, as shown in Figure 1, the sources on surface emit waves that will be refracted or reflected on the different layers composing the subsurface, and will eventually go back to receivers on the

978-1-4799-7023-0/14 $31.00 © 2014 IEEE
DOI 10.1109/WACCPD.2014.8

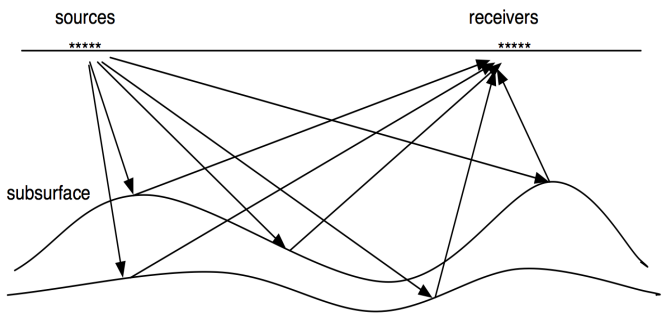


Figure 1: Seismic imaging

surface after a certain time. The receivers on subsurface will capture those waves with some devices like seismograms. The data from one source to one receiver is called a trace. The data from one source to multiple receivers is called a shot. Therefore, a shot includes many traces. The Kirchhoff Migration aims at generating a 3D-image of the subsurface by retrieving the position of the reflection points, therefore showing the different layers of the target. The key to this problem is to figure out how much time it takes for a wave to travel from a source to any point of coordinate (x, y, z) of the target, and back to the receiver. We can derive a set of (x, y, z) points in the subsurface that could be the potential reflection points. The candidates for the reflection points can be identified by analyzing the trace recording: it shows as a peak in the recordings. Therefore, out of all the (x, y, z) points of the target, only those who return a two-way traveltime that matches the peak of the trace are suitable to be the true reflection points we seek.

The real reflection points could be "candidates" for many traces. So if we repeat the seismic survey operation for a large number of traces, then the set of real reflection points will emerge from the set of "occasional" candidates that showed up on very few traces. In other words, each trace gives a contribution to the image. The closer the traveltime for a given (x, y, z) point is to the trace's peak, the more the trace rewards this point. This can be translated by the following formula, where the subscripts s and r refer to 'source' and 'receiver', respectively:

$$im(x, y, z) = \sum_{s} \sum_{r} A_{s,r}(x, y, z) \frac{dP(\tau_s(x, y, z) + \tau_r(x, y, z))}{dt}$$

At point (x, y, z), for a given trace, we take a sample from the recording, at time $\tau_s(x, y, z) + \tau_r(x, y, z)$: the closer it is to the actual peak time for the recording, the higher its value is. We ponderate this value by a weight function $A_{s,r}(x, y, z)$, that arbitrarily favorizes (x, y, z) points with less awkward reflection angles, and positions that are more likely to match the true reflection point. Basically, the value of the final image at point (x, y, z) is the sum of all the contributions of the traces, to this point. This sum is expressed by the double sum over all sources and receivers. The number of traces in recent surveys can sometimes be over 500 millions, which consists of several TBs of data, regarding the acquisition. This emphasizes the I/O and computational-expensive costs of the algorithm.

We need to compute the time for a wave to travel from a point at the surface to any point of the target, as well as other attributes. We emulate a wave by casting rays in all directions. The results of this computation is referred to as "Green Functions". The subsurface/target is modeled as a 3D-grid and green functions contain the traveltime for all the points of this 3D-grid. The sources and receivers on the surface are modeled by a 2D-grid. We have as many green functions as we have nodes on the grid modeling the surface. All of these grids are coarse, but the final image (the output of Kirchhoff Migration) needs to be a much finer 3D-grid, which can be done by either interpolation or extrapolation techniques.

So the input of Kirchhoff Migration contains acquisition which includes all seismic traces gathered from the surface, and green functions which include parameters of the rays on a coarse grid. The output of the application is a 3D-image of the target/subsurface. The whole application processes each trace in unit. Because the input data is very large which could be up to several hundred GBs or several TBs, the I/O operation consumes a large portions of time. To overcome this problem, the application is split into two tasks:

- Coarse Task: do all I/O regarding seismic acquisition and green functions and does computation on the whole coarse grid.
- Image Task: migrate the trace and does computation on a coarse grid to obtain a fine grid using interpolation or extrapolation.

The workflow of the application is shown in Figure 2. The path of the execution will diverge depending on whether it is coarse task or image task. These two tasks are executed by different MPI processes. The reason that the algorithm is designed like this is because there are large I/O operations in this application and we try to overlap the I/O operation and the computation as much as possible. So when the image task does the migration for one trace, the coarse task can do the I/O operation to read the input data of the next trace and green function and send to the image task. When the image task finishes its computation, the input data has arrived so that it does not need to read the data itself. The migration in the image task is the most compute intensive part in this application and that is the main part that will be parallelized in this paper.

III. GPU Architecture and OpenACC Directives

GPU architectures differ significantly from CPU. In this paper, GPU is always referred to **G**eneral **P**urpose **GPU**. Nvidia GPU hardware cores are organized into an array of Streaming Multiprocessors (SMs) and each SM contains

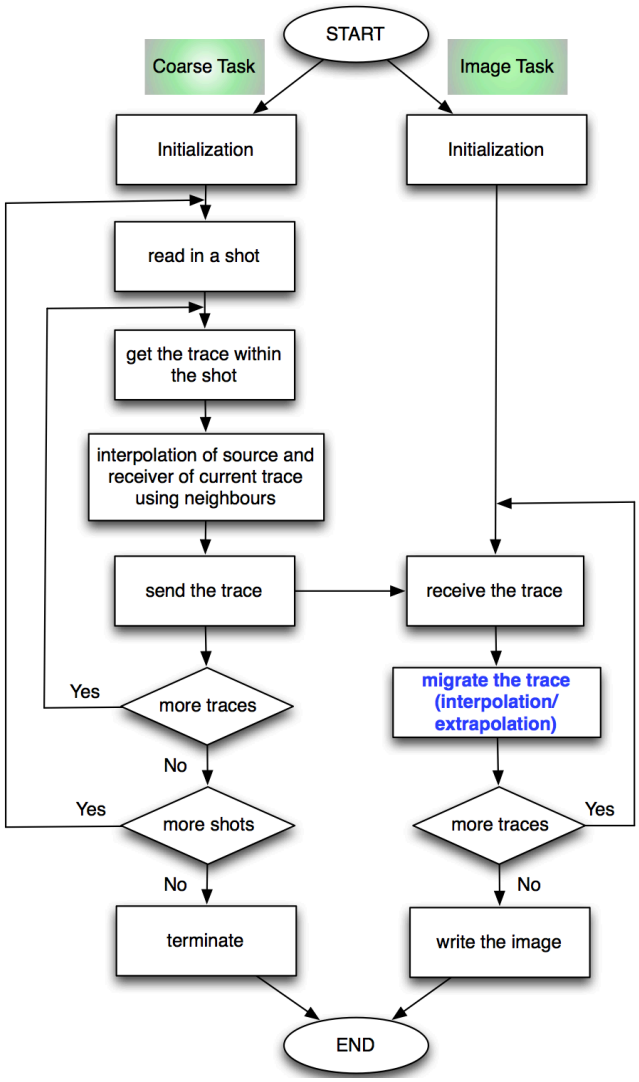


Figure 2: Application workflow

many cores named as Scalar Processors (SPs). Different number of threads can be launched to execute a computation kernel. These threads can be 1D, 2D or 3D thread blocks and each thread block could also contain 1D, 2D or 3D threads. One challenge for the programmers is how to effectively map the massive number of threads to the nested loops in their applications. Similar to CPU, GPU also has a complex memory hierarchy: the global memory which has the highest access latency; the shared memory, L1 cache and Read-Only Data Cache [5] which have very low latency; and the register which is the fastest part in GPU. So another challenge for the programmers is how to lay out data on different memory hierarchy to reduce the access latency and maximize coalesced memory access for all threads. Using low-level programming models such as CUDA and OpenCL to do these have been known as not only time consuming but also decreases code readability significantly. Directive-based high-level programming models for accelerators, e.g. OpenACC and OpenMP accelerator extensions, have been created to address GPU programmability challenges. This model allows the programmers to insert compiler directives into a program to offload the annotated code portion to accelerators. This approach heavily relies on compiler to generate the efficient code for thread mapping and data layout. This is more challenging to extract the optimal performance compared to other explicit programming models. However, this model can preserve the original code structure and parallelize the code incrementally thus easing the development effort. The execution model assumes that the main program runs on the host, while the compute intensive regions are offloaded to the attached accelerator. The memory spaces of the host and device are separate and they are not directly accessible from each other. To parallelize the compute kernels, OpenACC provides three levels of parallelism: the coarse-grain parallelism "gang", the fine-grain parallelism "worker" and vector parallelism "vector". The compiler decides how to map these parallelism to the GPU hardware. In most compiler implementations, `gang` is mapped to the thread block, `worker` mapped to the Y-dimension of each thread block and `vector` mapped to the X-dimension of the thread block. But how to parallelize a nested loop using these parallelism automatically and obtain the optimal performance is still challenging for compilers, since this requires the knowledge of the application and the target hardware. There are already a number of compilers that provides support for OpenACC. Those include PGI, CAPS and Cray, open-source OpenACC compilers include OpenUH [15], OpenARC [10] and accULL [12]. Most of these compilers except Cray compiler translate OpenACC code into CUDA code when the target platform is Nvidia GPU.

IV. Parallelization and Optimization Strategies

Section II has explained the workflow of Kirchhoff Migration. Aside from the communication between different processes, the most computation intensive part is the migration step which uses either interpolation or extrapolation techniques to obtain a fine grid from a coarse grid. Whether to use interpolation or extrapolation is controlled by the user in a configuration file. In the CPU serial code execution, the interpolation or extrapolation takes up to 99% - 100% computation cost of the migration step. So that is clearly the most time consuming portion. Both interpolation and extrapolation construct the unknown data points from some known data points. The difference is that in the interpolation, the unknown data points are between or within the known data points, whereas in the extrapolation, the unknown data points are outside the known data points. The grid before interpolation or extrapolation is called the coarse grid, and the grid after that is called fine grid.

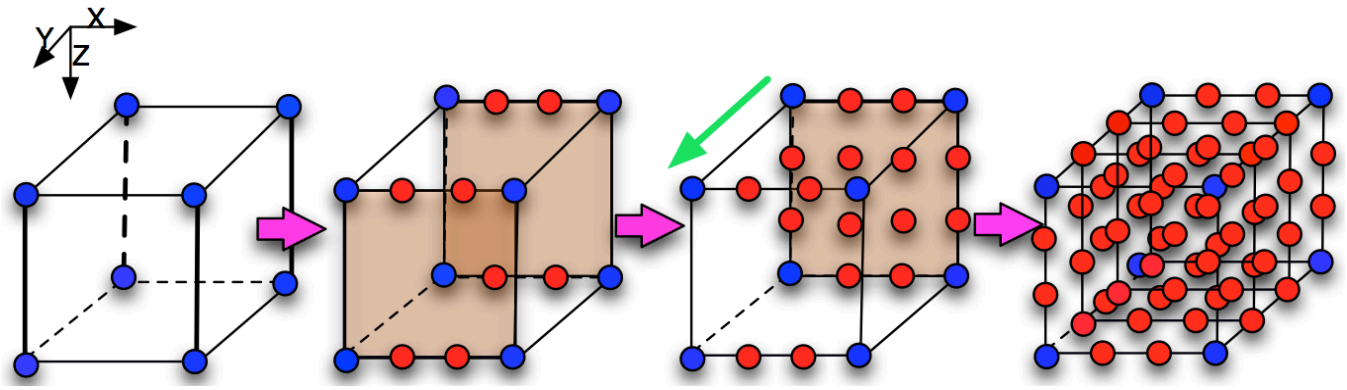


Figure 3: CPU Interpolation in Kirchhoff Migration

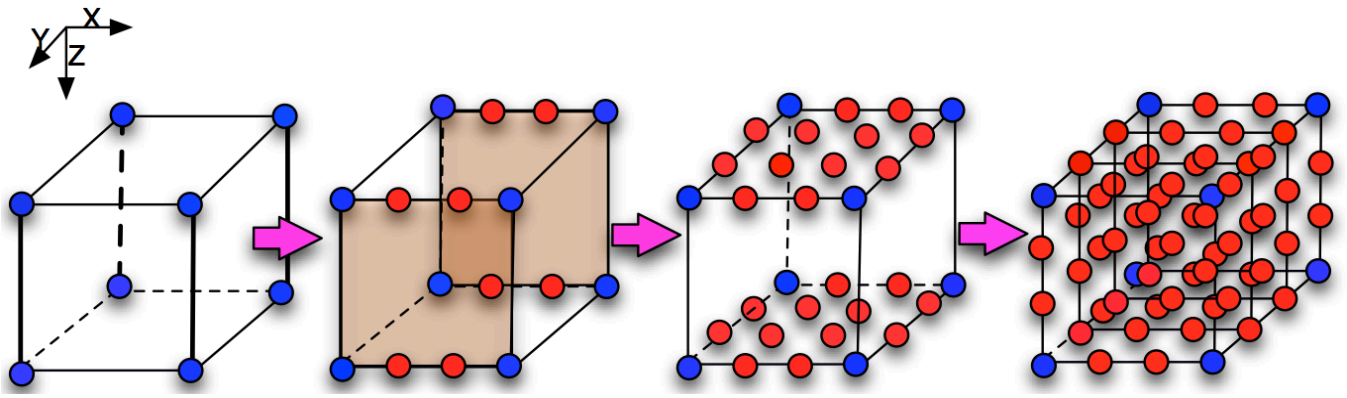


Figure 4: OpenACC Interpolation in Kirchhoff Migration

A. *Interpolation*

Figure 3 shows the CPU interpolation process in the Kirchhoff Migration. The eight blue points are known points at the beginning, and the goal is to construct all other unknown points in the grid. To get all the other unknown points, the algorithm works as follows:

- Access the 4 vertical "pillars" with the known 8 points from the coarse grid
- Perform interpolation in X-dimension in order to obtain the front plane and rear plane
- Perform the interpolation along Y-dimension. In each step of the Y-dimension interpolation, we first calculate the points on the top and bottom lines in that Y plane (or XZ plane) using the calculated data points from the front and rear planes, then interpolate all the points in Z-dimension in that plane.

Kirchhoff Migration was ported to CPUs using OpenMP. This optimized CPU port scaled linearly as the number of CPU threads increased. Listing 1 shows the Fortran CPU OpenMP pseudo code for the interpolation in Kirchhoff Migration. For CPUs, the data points on the top and bottom are first calculated in each XZ plane interpolation, then immediately used in the Z-dimension interpolation, so the cache utilization is efficient. This is because the data is firstly stored in Z-dimension, then in X and finally in Y-dimension and it is accessed in the same order. Since OpenMP is not the focus of this paper, we are not discussing the details in-depth.

```
!$OMP PARALLEL DO
do i=ixmin_c,ixmax_c
    do j=iymin_c,iymax_c
        ! get the 4 pillars from the coarse node
        do k
        enddo

        ! build the front and rear planes
        ! (interpolation in X-dimension)
        do m
        enddo

        ! build all planes in Y-dimension
        do l
            tmp_array(...) = ...
            ! interpolation in Z-dimension in each Y plane
            do k
               ... = tmp_array(...)
            enddo
        enddo
    enddo
enddo
!$OMP END PARALLEL DO
```

Listing 1: Interpolation OpenMP pseudo code in Kirchhoff Migration

However, the OpenMP port (using OpenMP 3.1 instead of OpenMP 4.0) is not efficient for the GPU architecture. The OpenMP port as-is is not best suited for the GPU architecture. For example, in the CPU implementation, the loop on Y-dimension interpolation is executed sequentially. All iterations in this loop are independent and therefore can be parallelized using OpenACC but array privatization technique [17] is needed for correct results. This is because in CPU implementation, each thread uses a temporary buffer to store the values of each XZ plane. When we parallelize the Y-dimension interpolation loop, the temporary buffer should be enlarged to hold all XZ planes, so that all GPU threads can write into different memory addresses without data race. Although OpenACC also provides "private" loop clause, the compiler implementation is not mature enough for privatizing arrays correctly, especially for complicated kernels.

After the arrays are privatized the kernel can be safely parallelized. In order to improve the performance, we can do better by accessing the data from the memory with lesser access latency. By default all data are stored in GPU global memory. In order to reduce the memory access latency, the data can be stored in the memory with lower latency. In GPU memory hierarchy, there is a new cache in Kepler called Read-Only Data Cache [5]. The requirement of using this cache is that the data need to be read only, then the compiler annotate those read only data with some CUDA intrinsic keywords and the hardware can automatically cache those data into read only cache. To take advantage of this cache, the interpolation computation needs to be reordered. The interpolation after reordering is shown in Figure 4. The first and second steps are the same as the CPU implementation, but in the third step, we do the interpolation in Y-dimension for all XZ planes and get all values on the top and bottom planes. The final step is to interpolate along Z-dimension to construct all XY planes between the top and bottom planes. The data on the top and bottom planes are read only now and they can be safely cached into read only data cache. The detailed code for the OpenACC implementation is shown in Listing 2. The loop fission is applied to the original single nested loop, thus make each interpolation step as a kernel. This is because unlike CPU code in which only the outermost loop is parallelized, the GPU architecture prefers small kernels and needs more parallelism to hide the memory access latency and in order to utilize read only cache. Although loop fission generates more kernels which leads to more kernel launch overhead, the benefits outweigh such trivial penalty significantly.

The number of points to be interpolated can be controlled in a configuration file. The smaller the distance between each point in the fine grid, the more points need to be interpolated, and the final migrated image is larger and more accurate. Note that the cube in Figure 3 is just one cube in the entire coarse grid. All the data points in the X-Y dimensions of the coarse grid need to interpolateand for each coarse node, the interpolation loops over the X-Y-Z dimensions of the fine grid. This illustrates the compute intensive property in interpolation.

```
!$ACC KERNELS LOOP
do i=ixmin_c,ixmax_c
    do j=iymin_c,iymax_c
        ! get the 4 pillars from the coarse node
        do k
        enddo
    enddo
enddo

!$ACC KERNELS LOOP
do i=ixmin_c,ixmax_c
    do j=iymin_c,iymax_c
        ! build the front and rear planes
        !(interpolation in X-dimension)
        do m
        enddo
    enddo
enddo

!$ACC KERNELS LOOP
do i=ixmin_c,ixmax_c
    do j=iymin_c,iymax_c
        ! build the top and bottom planes
        !(interpolation in Y-dimension)
        do l
            tmp_array(...) = ...
        enddo
    enddo
enddo

!$ACC KERNELS LOOP
do i=ixmin_c,ixmax_c
    do j=iymin_c,iymax_c
        ! build all planes between top and bottom planes
        do l
            !(interpolation in Z-dimension)
            do k
                ... = tmp_array(...)
            enddo
        enddo
    enddo
enddo
```

Listing 2: Interpolation OpenACC pseudo code in Kirchhoff Migration

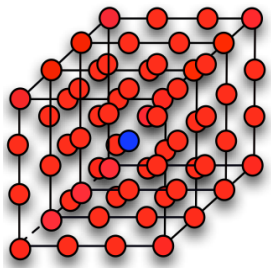


Figure 5: Extrapolation in Kirchhoff Migration

B. *Extrapolation*

Figure 5 shows the working mechanism of extrapolation for one coarse node. The node is in the center of the cube and the goal is to extrapolate all other unknown data points in the cube. The extrapolation is done using the first order

Taylor expansion:

$$f(x,y,z) = f(x_0,y_0,z_0) + \frac{\partial f}{\partial x}(x-x_0) + \frac{\partial f}{\partial y}(y-y_0) + \frac{\partial f}{\partial z}(z-z_0)$$

```
!$acc parallel loop collapse(2) gang
do iy = iymin_c, iymax_c
  do ix = iymin_c, iymax_c
    izmin_c = ...
    izmax_c = ...
    ......
    !$acc loop worker
    do iz = izmin_c, izmax_c
      ...
      cond = ...
      if(cond) cycle
      !$acc loop collapse(3) vector
      do iy_in = iymin_f, iymax_f
        do ix_in = ixmin_f, ixmax_f
          do iz_in = izmin_f, izmax_f
            !$acc atomic
               mig_extrap(iz_in,...) += ...
            !$acc end atomic
          enddo
        enddo
      enddo
    enddo
  enddo
enddo
```

Listing 3: Extrapolation OpenACC pseudo code in Kirchhoff Migration

The pseudo code for the extrapolation is shown in Listing 3. As we can see, the extrapolation algorithm loops over all coarse nodes in X-Y-Z dimensions, and for each coarse node, the extrapolation is performed in X-Y-Z dimension of the fine grid. To increase parallelism, we collapsed the innermost triple nested loop and distributed into vector parallelism, the Z-dimension of the coarse grid is distributed into worker parallelism. The Y-X dimensions of the coarse grid is done by gang parallelism. The reason that we use "atomic" directive to protect the write operation is that this algorithm has potential data race when it is parallelized. Figure 6 shows the detailed explanation about why there is a possibility of data race. In this figure, the colored data points are the points on the coarse grid. The point $(x0, y0+1, z0+1)$ extrapolates all the red space and the point $(x0+1, y0, z0)$ extrapolates all the blue space, and both the red and blue space overlap within a small cube. All the 8 data points shown in this figure overlap their extrapolation space in this cube. So whenever 8 data points extrapolate in parallel, the threads compete each other to write the data in the cube area. To prevent such data race, the "atomic" directive guarantees that each thread has exclusive access when writing in the cube area.

The extrapolation kernel in Kirchhoff Migration is a very large and complicated kernel. There are two kinds of extrapolations in this application called KIMONO and TAMONO. The profiling results show that KIMONO used 85 registers per thread and TAMONO used 215 registers per thread. The occupancy, which is the indicator of Thread Level Parallelism (TLP) in CUDA, for these two extrapolation kernels are 31% and 13%, respectively. These occupancy

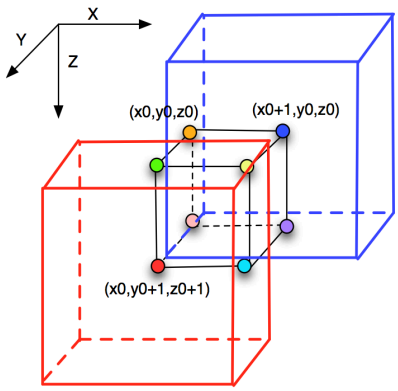


Figure 6: Potential data race in extrapolation. The extrapolation space of these 8 data points overlap within the central cube. To make the figure clear, only the extrapolation space of the upper right and lower left two points are drawn.

values indicate that there is no enough parallelism for these two kernels. This is obvious since the number of registers is limited in each SM of a GPU. So the more registers each thread uses, the lesser threads will run concurrently on each SM. Although high occupancy does not guarantee high performance, it is one of the factors contributing to the performance. Work in [1] demonstrates experiments using techniques such as increasing Instruction Level Parallelism (ILP) and processing multiple data per thread, thus the occupancy goes down and the performance improves. The author's conclusion was that low occupancy could also lead to high performance. However, The author did not try to keep those factors fixed and try other techniques to increase the occupancy and see whether that will improve the performance. In our OpenACC extrapolation kernel, we have already reordered the instructions to remove or reduce the dependences between instructions, thus increase ILP. By checking the generated CUDA kernel, we observed that the technique that processing multiple data per thread has been done by the compiler automatically. Now the occupancy is low and the performance is still poor. What we want to try is to increase the occupancy and see whether the performance improves.

In our extrapolation kernels, when the number of threads is fixed, reducing the number of registers per thread will improve the occupancy. This is based on the result from the CUDA occupancy calculator [4]. The result of reducing the number of registers per thread is register spilling which spills registers into L1 cache [3]. Therefore we cannot spill too many registers since the access latency from L1 cache is higher than register. However, zero register spilling leads to low occupancy, so we need to balance the cost of occupancy and register spilling. Figure 7 shows performance and occupancy results while using different number of registers. The result has verified our thoughts that appropriate occupancy

and register spilling are key to the performance improvement in the extrapolation kernel. The best number of registers per thread for KIMONO extrapolation and TAMONO extrapolation are 40 and 64, respectively. We use these best register numbers in our later experiments.

C. Asynchronous Data Transfer

Although in GPU computing, the data transfer between CPU and GPU is a serious issue for many applications, this is not a big issue in Kirchhoff Migration. For all the data to be used on GPU device, the device memory are allocated at the beginning of the application by using "`enter data`" directive and freed at the end of the application by using "`exit data`" directive. For the data movement in the middle of the computation, once the image task has received the green function and trace from the coarse task, we issue the data update request immediately by using "`update async`" directive so that the data is transferred from CPU to GPU asynchronously. Then the CPU performs the FFT operation for the trace data and we also transfer the result asynchronously to GPU. Here asynchronous data transfer of green function and FFT computation could overlap each other. Since the data transfer only takes a tiny fraction of the whole application, this does not impact the performance too much.

V. Preliminary Evaluation

The experiment platform is Cray cluster. Each node of the cluster has 20 cores (2 sockets) Intel Xeon E5-2680 v2 x86_64 CPU (Ivy Bridge) with 64GB main memory, and an Nvidia Kepler GPU card (K40) with 8GB main memory. We use PGI 14.6 compiler to evaluate the performance of Kirchhoff Migration application for both CPU code and OpenACC code. The CPU code runs on one socket that uses 10 OpenMP threads. Optimizationi flag used is -O3 for both OpenMP and OpenACC. CUDA version used by the OpenACC compiler is CUDA 5.5. The OpenACC code is compiled and linked with the flag "-ta=tesla:cc35,pin,nofma", in order to generate the code specific to Kepler architecture, use pinned memory for asynchronous data transfer, and disable FMA [16] to easily compare the OpenACC result with CPU result. We use one MPI process for coarse task and another one for image task in both CPU and OpenACC codes.

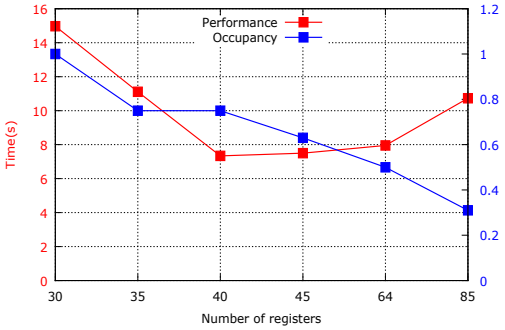
Two datasets are used in our experiment: ONELAYER and SALT. The ONELAYER data features a single horizontal reflector, transition between the two different layers at around z=1600m. The SALT dataset shows a salt dome. To maximize the GPU memory usage, we choose the smallest possible image resolution which is the step size in the fine grid. As a result, the image resolution is dx=dy=dz=5 for ONELAYER and dx=dy=dz=10 for SALT. The smallest compute unit is 1 shot. In order to get enough computation and avoid too long execution time, for ONELAYER we used 50 shots for interpolation and 4 shots for extrapolation, and for SALT we used only 1 shot for both interpolation and extrapolation.

There are three types of interpolation: offset gather, angle gather and angle gathers. They all follow the interpolation process as described in Section IV. The difference lies in the slight different computation part. The extraplation also has two types: KIMONO and TAMONO. They both follow the extrapolation algorithm in Section IV but TAMONO code structure is more complicated and has more computation. The experiment results for ONELAYER dataset is shown in Table I. The time taken for computation, communication and the whole application is measured. The application does not include the final image writing time since that takes long time to write several GBs image data and non-parallelizable. Parallel I/O could be used, but we will not focus on that in this paper. The computation part of the application is the migration part which includes interpolation/extrapolation ($\geq$99% of migration time) and some other computation like FFT operation.

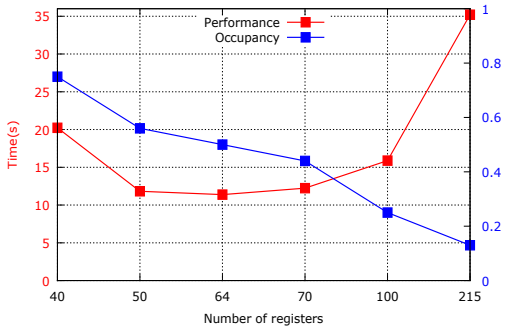
For the offset gather interpolation, we got a speedup of 11.42x. For the whole application, a speedup of 6.31 is achieved. The speedup of the whole application seem lesser than the computation speedup, this is due to communication and I/O cost. The communication is required in this application since the coarse task needs to send the trace and green functions to the image task. When the computation finishes too fast, the input data to the image task may have not arrived yet so that the image task spends more time to wait for the data. This observation is true for all types of interpolation. Also we notice that the application time is longer than the sum of the computation and communication time. This time gap is the I/O operation of the coarse task. This means although we try to use two processes to handle I/O and computation differently, the I/O and computation still could not completely overlap each other. Especially when the computation is faster, such gap will be larger. Note that only the I/O in the coarse task and computation in the image task can overlap each other, the communication between the coarse task and image task cannot overlap either the I/O or the computation. These conclusions are true for all types of interpolations.

It is observed that the angle gather/gathers speedup is higher than the offset gather speedup. This is because there are "acos()" operation in these kernels. GPU has Special Function Units (SFU) on the hardware for "acos" like transcendental functions. However, CPU only has software solution for these functions.

Table II shows the results of SALT dataset. We can see that SALT got less speedup than ONELAYER in interpolation. This is because the amount of computation in SALT is less than ONELAYER. In our experiments, with SALT data, the number of coarse nodes in the coarse grid is 134x135x40, and the image resolution is 10x10x10, the number of fine nodes in the fine grid is the product of them which is



(a) KIMONO Extrapolation



(b) TAMONO Extrapolation

Figure 7: Performance and occupancy with changing registers per thread. In both figures, on the right side, the number of registers per thread is high which caused low occupancy which further leads to low performance. On the left side, the number of registers per thread is low which caused high occupancy, but more register spilling caused low performance. The middle part has moderate occupancy and low register spilling which leads to high performance.

Table I: Kirchhoff Migration performance and speedup for ONELAYER dataset

Migration Technique	Type	Version	Computation		Communication	Application	
			Time(s)	Speedup	Time(s)	Time(s)	Speedup
Interpolation	offset gather	CPU (10 cores)	50.46	1	0.4	53.33	1
		OpenACC	4.42	**11.42**	1.72	8.45	**6.31**
	angle gather	CPU (10 cores)	115.29	1	0.4	118.18	1
		OpenACC	6.07	**18.99**	0.55	9.1	**12.99**
	angle gathers	CPU (10 cores)	136.2	1	0.4	139.25	1
		OpenACC	6.63	**20.54**	0.52	9.62	**14.48**
Extrapolation	KIMONO	CPU (10 cores)	6.96	1	0.06	9.16	1
		OpenACC	2.57	**2.71**	0.05	4.34	**2.11**

Table II: Kirchhoff Migration performance and speedup for SALT dataset

Migration Technique	Type	Version	Computation		Communication	Application	
			Time(s)	Speedup	Time(s)	Time(s)	Speedup
Interpolation	offset gather	CPU (10 cores)	5809.59	1	334.8	6612.28	1
		OpenACC	96.09	**6.05**	461.84	1892.22	**3.49**
	angle gather	CPU (10 cores)	12814.14	1	332.34	13612.84	1
		OpenACC	1174.18	**10.91**	442.86	2081.61	**6.54**
	angle gathers	CPU (10 cores)	15041.34	1	333.51	15842.53	1
		OpenACC	1261.33	**11.92**	444.52	2166.53	**7.31**
Extrapolation	KIMONO	CPU (10 cores)	6922.19	1	331.36	7719.02	1
		OpenACC	2701.33	**2.56**	441.25	3590.02	**2.15**
	TAMONO	CPU (10 cores)	23494.76	1	330.34	24292.81	1
		OpenACC	4581.60	**5.13**	439.16	5462.1	**4.45**
		OpenACC_UH	3498.82	**6.72**	437.97	4382.47	**5.54**

723600000. While in ONELAYER, the number of coarse nodes is 72x72x30, and the image resolution is 5x5x5, so the number of fine nodes in the fine grid is 1244160000 which is close to twice the size of SALT data. Based on Amdahl's law, when the parallelizable computation is larger, we can get higher speedup when the same number of processors are used.

Implementation of OpenACC features differ from one compiler to another. Using more than one compiler will allow us to study how different implementations can contribute to optimal performance. To compare the performance differences, we also tried our open source OpenACC compiler called OpenUH [15] that uses a different kernel transformation technique than PGI compiler. Due to the kernel complexity and time limit, we only compared the performance of TAMONO extrapolation. The performance and speedup are shown in OpenACC_UH row in Table II. We can see that the code using OpenUH kernel transformation technique was around 18 minutes faster and therefore achieved higher speedup than PGI. The reason for such performance differences is due to the OpenACC kernels' different code generation.

```
et4 = iyxmax_c-iyxmin_c+1
tc3 = izmax_c-izmin_c+1

j1651 = 0;
BB_22:;
i_3 = blockIdx.x-et4 + j1651;
if(i_3>=0) goto BB_28;
if(threadIdx.x+threadIdx.y!=0)
    goto BB_36;
! gang work
gang_list = ...
shared_gang_list = gang_list;
__threadfence_block();

BB_36:;
__syncthreads();
gang_list = shared_gang_list;

j1649 = 0;
BB_35:;
if((threadIdx.y-tc3+j1649) >= 0) goto BB_41;
if(threadIdx.x!=0) goto BB_124;
! worker work before cycle
cond = ...
worker_list1 = ...
e284[threadIdx.y] = cond;
shared_worker_list1[threadIdx.y] = worker_list1;
__threadfence_block();
BB_124:
worker_list1 = shared_worker_list1[threadIdx.y];
if(e284[threadIdx.y] != 0) goto BB_41;
if(threadIdx.x!= 0) goto BB_110;
! worker work after cycle
shared_worker_list2[threadIdx.y] = worker_list2;
__threadfence_block();
BB_110:;
worker_list2 = shared_worker_list2[threadIdx.y];
   .
i2224s = 0;
x119 = iymax_f-iymin_f+1;
x117 = izmax_f-izmin_f+1;
x118 = ixmax_f-ixmin_f+1;
et412 = x118*x117*x119;
et422 = threadIdx.x - et412;
BB_104:;
i_1 = threadIdx.x + i2224s;
i_5 = et422 + i2224s;
if(i_5 >= 0) goto BB_107:
   ! vector work
   ...
BB_107:;
i2224s = i2224s + 32;
if((i2224-et412)<0) goto BB_104;

BB_41:;
j1649 = j1649 + blockDim.y;
if((j1649-tc3)<0) goto BB_35;

BB_28:;
j1651= j1651 + gridDim.x
if((j1651-et4)<0) goto BB_22
```

Listing 4: PGI kernel transformation

```
iy_size = iymax_c-iymin_c+1;
ix_size = ixmax_c-ixmin_c+1;
iyx = blockIdx.x;
while(iyx<iy_size*ix_size)
{
    iy = ...; ix=...;
    ...
    iz = threadIdx.y + izmin_c;
    while(iz<izmax_c)
    {
        cond = ...
        if(cond) goto IZ_END;

        size1 = iymax_f-iymin_f+1;
        size2 = ixmax_f-ixmin_f+1;
        size3 = izmax_f-izmin_f+1;
        iyxz_in = threadIdx.x;
        while(iyxz_in<size1*size2*size3)
        {
            iy_in=...;ix_in=...;iz_in=...;
            ...
            iyxz_in += blockDim.x;
        }
        IZ_END:;
        iz += blockDim.y;
    }
    iyx += gridDim.x;
}
```

Listing 5: OpenUH kernel transformation

For the kernel in Listing 3, PGI and OpenUH translate them differently in Listing 4 and Listing 5, respectively. For the kernel using `gang`, `worker` and `vector` parallelism, PGI uses the first thread in each thread block (threadIdx.x+threadIdx.y==0) to do the gang work, then it stores the results into the shared memory and broadcast to other threads in the thread block. In the next worker loop, first each worker fetches the gang loop results from the shared memory, then the first thread in each worker (threadIdx.x==0) does the worker work. When there is a cycle statement in the worker loop, that worker first stores the results before the cycle statement and broadcast to other threads, and whoever workers passing the cycle statement fetches those results before cycle from the shared memory. All the vector threads in each worker does the work in the vector loop. OpenUH compiler does not use this type of kernel transformation. Instead it lets all workers inside each gang do the gang work redundantly, and all the vector threads inside each worker do the worker work redundantly, so that there is no operations to store the temporary gang and worker results into shared memory and broadcast to other threads and then fetch those results from the shared memory again. This different in implementation may be a possible reason for the performance differences in our extrapolation experiment.

VI. Related Work

Kirchhoff Migration is one of the most widely used seismic migration methods. There are different implementations of this application and different programming models have been used to accelerate this application. Shi et al. [13] used CUDA to parallelize this application and used CUDA streams to overlap the data communication and computation, and they also discussed the floating point result difference between GPU and CPU. Due to the implementation difference, they did not have interpolation or extrapolation in their migration. Because they did not use the design like us to split the I/O operation and computation, their data transfer dominates the execution time, so they had to use CUDA streams to overlap the data transfer and computation. Our algorithm design allows us to focus on the computation part, but we still used asynchronous data transfer to transfer the green function and trace buffer to

the GPU to further minimize the data movement cost. We used OpenACC directives instead of CUDA streams which greatly improved the productivity.

Panetta et al. [11] ported the Kirchhoff Migration on a cluster of GPUs. They used Tesla C1060 GPU which does not have Read-Only Data Cache, so they parallelized the migration with CUDA using global memory without considering to fetch the data from a memory with less access latency. And their migration only has interpolation, without extrapolation. In addition, since they ported to a cluster, they also discussed the load balancing issue. Sun et al. [14] used OpenCL programming model to parallelize Kirchhoff Migration and applied vector operation and texture memory to optimize the performance, and finally obtained the comparable performance to CUDA code.

VII. Conclusion

This paper briefly introduces the widely used Kirchhoff Migration application within the Oil and Gas industry. We then discuss OpenACC's usability for this application. We particularly deal with the interpolation and extrapolation of the migration computation part. We discuss the following techniques to optimize the application port:

- Ways to reorder the computation in interpolation to optimize the memory access,
- Use the atomic operation to prevent the potential data race in extrapolation,
- Choose the optimal number of registers per thread to balance the occupancy and register spilling to optimize the extrapolation kernel performance.

We demonstrate our findings using ONELAYER and SALT two datasets, and analyze the performance difference. We also analyze different kernel transformation techniques from OpenACC to CUDA in both PGI and OpenUH compilers, and compared their performances in one extrapolation type. As future work, we plan to investigate new techniques to further reduce the communication cost between the coarse task and image task and use multi-GPU to further improve the performance.

Acknowledgment

The authors would like to thank PGI compiler team for providing technical support and TOTAL for providing the computing resources.

References

[1] Better Performance at Lower Occupancy. http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf, 2014.

[2] CUDA. http://www.nvidia.com/object/cuda_home_new.html, 2014.

[3] CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2014.

[4] CUDA Occupancy Calculator. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#calculating-occupancy, 2014.

[5] Kepler Tuning Guide. http://docs.nvidia.com/cuda/kepler-tuning-guide/, 2014.

[6] OpenACC. http://www.openacc-standard.org, 2014.

[7] OpenCL Standard. http://www.khronos.org/opencl, 2014.

[8] OpenMP. www.openmp.org, 2014.

[9] Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP: A Hybrid Multi-core Parallel Programming Environment. In *Workshop on GPGPU*, 2007.

[10] Seyong Lee, Dong Li, and Jeffrey S Vetter. Interactive Program Debugging and Optimization for Directive-Based, Efficient GPU Computing, 2014.

[11] Jairo Panetta, Thiago Teixeira, Paulo RP de Souza Filho, Carlos A da Cunha Finho, David Sotelo, F Da Motta, Silvio Sinedino Pinheiro, I Pedrosa, Andre L Romanelli Rosa, Luiz R Monnerat, et al. Accelerating Kirchhoff Migration by CPU and GPU Cooperation. In *Computer Architecture and High Performance Computing, 2009. SBAC-PAD'09. 21st International Symposium on*, pages 26–32. IEEE, 2009.

[12] Ruymán Reyes, Iván López-Rodríguez, Juan J Fumero, and Francisco de Sande. accULL: An OpenACC Implementation with CUDA and OpenCL Support. In *Euro-Par 2012 Parallel Processing*, pages 871–882. Springer, 2012.

[13] Xiaohua Shi, Chuang Li, Shihu Wang, and Xu Wang. Computing Prestack Kirchhoff Time Migration on General Purpose GPU. *Computers & Geosciences*, 37(10):1702–1710, 2011.

[14] Peiyuan Sun and Xiaohua Shi. An OpenCL Approach of Prestack Kirchhoff Time Migration Algorithm on General Purpose GPU. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2012 13th International Conference on*, pages 179–183. IEEE, 2012.

[15] Xiaonan Tian, Rengan Xu, Yonghong Yan, Zhifeng Yun, Sunita Chandrasekaran, and Barbara Chapman. Compiling A High-Level Directive-based Programming Model for GPGPUs. In *Intl. workshop on LCPC 2013*, pages 105–120. Springer International Publishing, 2014.

[16] Nathan Whitehead and Alex Fit-Florea. Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. *rn (A+ B)*, 21:1–1874919424, 2011.

[17] Rengan Xu, Xiaonan Tian, Sunita Chandrasekaran, Yonghong Yan, and Barbara Chapman. NAS Parallel Benchmarks on GPGPUs using a Directive-based Programming Model. In *Intl. workshop on LCPC 2014*, 2014.

[18] Özdogan Yilmaz and Stephen M Doherty. *Seismic Data Processing*, volume 2. Society of Exploration Geophysicists Tulsa, 1987.