Compiling a High-level Directive-Based Programming Model for GPGPUs

Xiaonan Tian, Rengan Xu, Yonghong Yan, Zhifeng Yun, Sunita Chandrasekaran, and Barbara Chapman

Department of Computer Science, University of Houston
Houston TX, 77004 USA
{xtian2,rxu6,yyan3,zyun,schandrasekaran,bchapman}@uh.edu

Abstract. OpenACC is an emerging directive-based programming model for programming accelerators that typically enable non-expert programmers to achieve portable and productive performance of their applications. In this paper, we present the research and development challenges, and our solutions to create an open-source OpenACC compiler in a main stream compiler framework (OpenUH of a branch of Open64). We discuss in details our loop mapping techniques, i.e. how to distribute loop iterations over the GPGPU's threading architectures, as well as their impacts on performance. The runtime support of this programming model are also presented. The compiler was evaluated with several commonly used benchmarks, and delivered similar performance to those obtained using a commercial compiler. We hope this implementation to serve as compiler infrastructure for researchers to explore advanced compiler techniques, to extend OpenACC to other programming languages, or to build performance tools used with OpenACC programs.

1 Introduction

Computational accelerators that provide massive parallelism such as NVIDIA GPGPUs and Intel Xeon Phi, or those that provide special-purpose application engines such as DSP have become viable solutions to build high performance supercomputers, as well as special-purpose embedded systems. However, one of the critical challenges to fully exploit the hardware computation capabilities is the need for productive programming models. OpenCL and CUDA are widely-used low-level programming models designed for programming GPGPUs. These two programming models require rewriting of most of the application program from its CPU version that users want to offload to accelerators. This has been known to be a non-productive approach.

OpenACC [5] is an emerging standard for programming accelerators in heterogeneous systems. The model allows developers to mark regions of code for acceleration in a vendor-neutral manner. It is built on top of prior efforts adopted by several compiler vendors (notably PGI and CAPS Enterprise). OpenACC is intended to enable programmers to easily develop portable applications to

maximize performance and power efficiency of the hybrid CPU/GPU architecture. Compiler implementaiton and enhancement in the model are underway by several industry compilers, notably from Cray, PGI and CAPS. However, their source codes are mostly inaccessible to researchers and they cannot be used to gain an understanding of the OpenACC compiler technology or to explore possible improvements and suggest language extensions to the model.

In this paper, we present our experience of constructing an OpenACC compiler in the OpenUH open source compiler framework [10], with goals to enable a broader community participation and dialog related to this programming model and the compiler techniques to support it. We also hope this implementation to serve as compiler infrastructure for researchers that are interested in improving OpenACC, extending the OpenACC model to other programming languages, or building tools that support development of OpenACC programs. Specifically, the features of the compiler and our contributions are summarized as follows:

1. We constructed a prototype open-source OpenACC compiler based on a branch of main stream Open64 compiler. Thus the experiences could be applicable to other compiler implementation efforts.
2. We provide multiple loop mapping strategies in the compiler on how to efficiently distribute parallel loops to the threading architectures of GPGPU accelerators. Our findings provide guidance for users to adopt suitable loop mappings depending on their application characteristics.
3. OpenUH compiler adopts a source-to-source approach and generates readable CUDA source code for GPGPUs. This gives users opportunities to understand how the loop mapping mechanism are applied and to further optimize the code manually. It also allows us to leverage the advanced optimization features in the backend compilation step by the CUDA compiler.

We evaluate our compiler with several commonly used benchmarks, and showed the similar performance results to those obtained using a commercial compiler. The remainder of this paper is organized as follows: Section 2 gives an overview of OpenACC model. Section 3 presents implementation details of the OpenACC compiler. Section 4 shows the detail of runtime support. Section 5 discusses the results and evaluation. Section 6 provides a review of the related work. Conclusion and future work are presented in Section 7.

2 Overview of OpenACC Programming Model

OpenACC is a high-level programming model that can be used to port existing HPC applications on different types of accelerators with minimum amount of effort. It provides directives, runtime routines and environment variables as its programming interfaces. The execution model assumes that the main program runs on the host, while the compute-intensive regions of the program are offloaded to the attached accelerator. The accelerator and the host have separate memory, and the data movement between them need to be handled explicitly. OpenACC provides different types of data transfer clauses and runtime call in

its standard. To reduce the performance impacts of data transfer latency, OpenACC also allows asynchronous data transfer and asynchronous computation with the CPU code to enable overlapping of data movement and computation.

Figure 1 shows a simple OpenACC vector addition exampe. The `acc data` directive, which identifies a data region, will create `a` and `b` in device memory and then copy the respective data into device at the beginning of the data region. The array `c` will be copied out after finishing the code segment of the region. The `acc kernels` directive means the following block is to be executed on device. The `acc loop` directive causes the distributions of loop iterations among the threads on the device.

```
#pragma acc data copyin( a[0:n], b[0:n] ), copyout( c[0:n] )
{
  #pragma acc kernels
  {
    #pragma acc loop independent
    for ( i =  0;  i < n; i++ ) {
      c[i] = a[i] + b[i];
    }
  }
}
```

Fig. 1: OpenACC vector addition example

3 Compiler Implementation

The creation of an OpenACC compiler requires both innovative research solutions to the challenges of mapping high-level loop iterations to low-level threading architectures of the hardware, and also large amount of engineering work in compiler development to handle parsing, transformation and code generations. It also requires runtime support for handling data movement and scheduling of computation on the accelerators. The compiler framework we are using is OpenUH compiler, a branch of the open source Open64 compiler suite. Figure 2 shows the components of the OpenUH framework. The compiler is implemented in highly component-oriented way and composed of several modules, each of which operates on a multi-level IR called WHIRL. From top, each module translates the current level of WHIRL to its lower-level form.

We have identified the following challenges that must be addressed to create an OpenACC implementation. First, it is very important that we create an extensible parsing and IR systems to facilitate addition of new features of future language revisions and to support aggressive compiler transformation and optimizations. Fortunately, the extensibility of OpenUH framework and WHIRL IR allow us to easily add those extensions with decent amount of work. Secondly, we need to design and implement an effective means for the distribution of loopnest across thread hierarchy of GPGPUs. We discuss in more details of our solutions in section 3.1. Thirdly, we need to create a portable runtime to support data handling, reductions operations, and GPU kernel launching. Runtime support will be discussed in more details in Section 4.

We decide to use the source-to-source approach, as shown in Figure 2. WHIRL2C tool has been enhanced to output compilable C program from the CPU portion of the original OpenACC code, and we have created a WHIRL2CUDA tool that

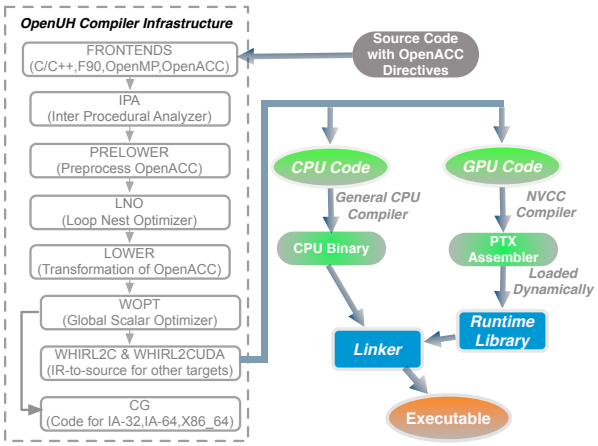


Fig. 2: OpenUH compiler framework for OpenACC

will produce NVIDIA CUDA kernels after the transformation of offloading code regions. Compared to binary code generation, the source-to-source approach gives much more flexibilities to users. It allows to leverage advanced optimization features in the backend compilation step by nvcc. It also gives user options to manually optimize the generated CUDA code for further performance improvement.

3.1 Loop Transformation

Programmers usually offload the computation intensive loopnest to massive parallel accelerators. One of the major challenges of compiler transformation is to create a uniformed loop distribution mechanism that can effectively map loopnest iteration across the GPU parallel system. As an example, NVidia GPGPUs has two level of parallelisms: block-level and thread-level. Blocks can be organized as multi-dimensional in a grid and threads in a block can also be multi-dimensional. How to distribute iterations of multi-level loopnest across the multi-dimensional blocks and threads is a nontrivial problem.

OpenACC provides three level of parallelisms for mapping loop iterations to the accelerators' thread structures: coarse grain parallelism "gang", fine grain parallelism "worker" and vector parallelism "vector". OpenACC standard gives the flexibility of interpreting them to the compiler. For NVIDIA GPU, some compilers map each gang to a thread block, and vector to threads in a block and ignore worker [6]; other compilers map gang to the x-dimension of a grid block, worker to the y-dimension of a thread block, and vector to the x-dimension of a thread block [2]. There are also compilers that map each gang to a thread block, worker to warp and vector to SIMT group of threads [7].

In our implementation, we evaluated 8 loopnest mapping algorithms covering single loop, double nested loop, and triple nested loop as shown in Figures 3,

6, 7. If the depth of the nested loop is more than 3, the OpenACC `collapse` clause will be used. More specifically, gangs are mapped to blocks and vectors are mapped to threads in each block. Both gang and vector can be multi-dimensional. The worker clause is omitted in current OpenUH compiler. Table 1 shows the mapping terminology we used between OpenACC and CUDA.

Table 1: OpenACC and CUDA Terminology Mapping

OpenACC clause	CUDA	Comment
gang (integer expression)	block	If there is an integer expression for this gang clause, it defines the number of blocks in one dimension of grid.
vector (integer expression)	thread	If there is an integer expression for this vector clause, it defines the number of threads in one dimension of block.

Memory coalescing is an important part that needs to take into careful consideration in compiler loop transformation. Different mapping can heavily affect the application's performance. Adjacent threads (in x-dimension of a block) taking consistent memory space can improve performance. Therefore we need to make sure the loop iteration mapped to vector x-dimension operates the continuous memory operands. The single loop and the inner loop iteration in double nested loop are mapped to x-dimension of threads. For triple nested loop, we selected three examples that are typically encountered in the OpenACC program. We mapped the innermost loop of Map3_1 and Map3_2 to operate on the continuous memory, but in Map3_3 it is mapped to the outmost loop to compute continuous memory. The reason of this mapping for Map3_3 is because we have a particular stencil application requiring the pattern likes this.

Single Loop. N iterations are equally distributed among gangs and vectors. Both gang and vector are one dimension. It means the grid and thread-block are also one dimension. Each thread takes one iteration at a time and then moves ahead with $blockDim.x * gridDim.x$ stride. Figure 3 show the mapping and transformation for this single loop.

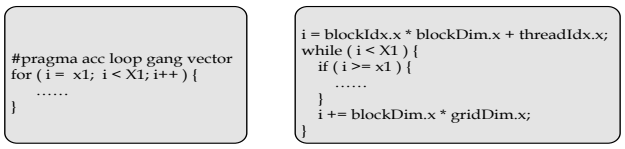


Fig. 3: One loop transformation.

Double Nested Loop. Figure 4 shows the double nested loop iteration distribution across gangs and vectors. The red one means current working area, green one means the finished computation, and white means untapped. The axises i and j represent the outer and inner loop iterations. Figure 4(a) shows the first working area, and the next status is in Figure 4(b). After finishing the last one in j axis, working area moves ahead into another i iteration. The

computation is not finished until all the rectangles turn to be green. The stride (length of the rectangle) in i and j depends on different mapping algorithms.

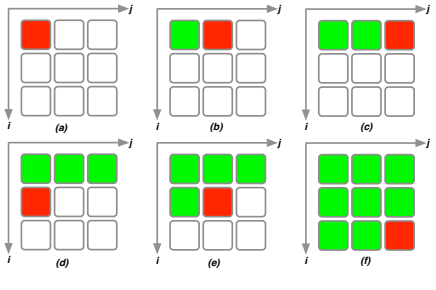


Fig. 4: Double nested loop iteration distribution.

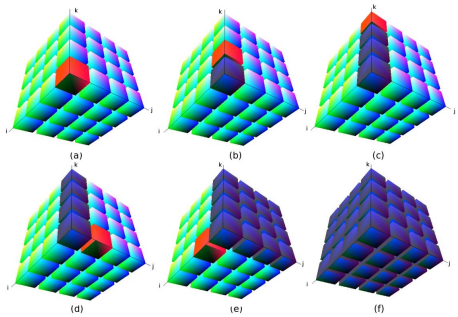


Fig. 5: Triple nested loop iteration distribution.

There are four different double nested loop cases, and the mapping algorithms are different from each other. Figure 6 shows the mapping for each case:

- Both gangs and vector are one dimension. The outer loop is distributed across the gang and inner loop is executed among threads in each gang. The stride in i and j axis are $gridDim.x$ and $blockDim.x$. The translated CUDA code from this case is shown in Figure 6 Map2_1.
- One dimensional gang, two dimensional vector. After the mapping, the outer loop stride is $gridDim.x{*}blockDim.y$ and the inner loop stride is $blockDim.x$. The translated CUDA code is shown in Figure 6 Map2_2.
- Two dimensional gangs and one dimensional vectors. After the mapping, the outer loop stride is $gridDim.y$ and the inner loop stride is $gridDim.x * blockDim.x$. The translated CUDA code is shown in Figure 6 Map2_3.
- Both grid and block are two dimensions. After the mapping, the outer loop stride is $gridDim.y * blockDim.y$ and the inner loop stride is $gridDim.x * blockDim.x$. The translated CUDA code is shown in Figure 6 Map2_4.

Triple Nested Loop. Figure 5 shows triple nested loop iteration distribution across gangs and vectors. In this figure, the red one means current working area, blue one means the finished computation, and green means untapped. The axises i, j, and k represent the outermost, middle and innermost loop iterations. At the first step, GPU takes computation from axis k in Figure 5(a). When finishing, working area moves ahead along the k axis (5(b)) until all the computation in k axis is done (5(c)). After this, the computation will move to the next j (5(d)). Repeat the first step until j reaches the boundary. Once all the computation on the j, k space are done, i moves a stride ahead, and reset j, k axises (5(e)). The computation repeats until all the computation is done (5(f)).

For the three different triple nested loops, Figure 7 shows the mapping:

- Both gang and vector are two dimensional. After the mapping, the outmost loop stride in i, j, k axises are $griddim.x$, $blockDim.y * griddim.y$ and $blockDim.x$. The translated CUDA code is shown in Figure 7 Map3_1.

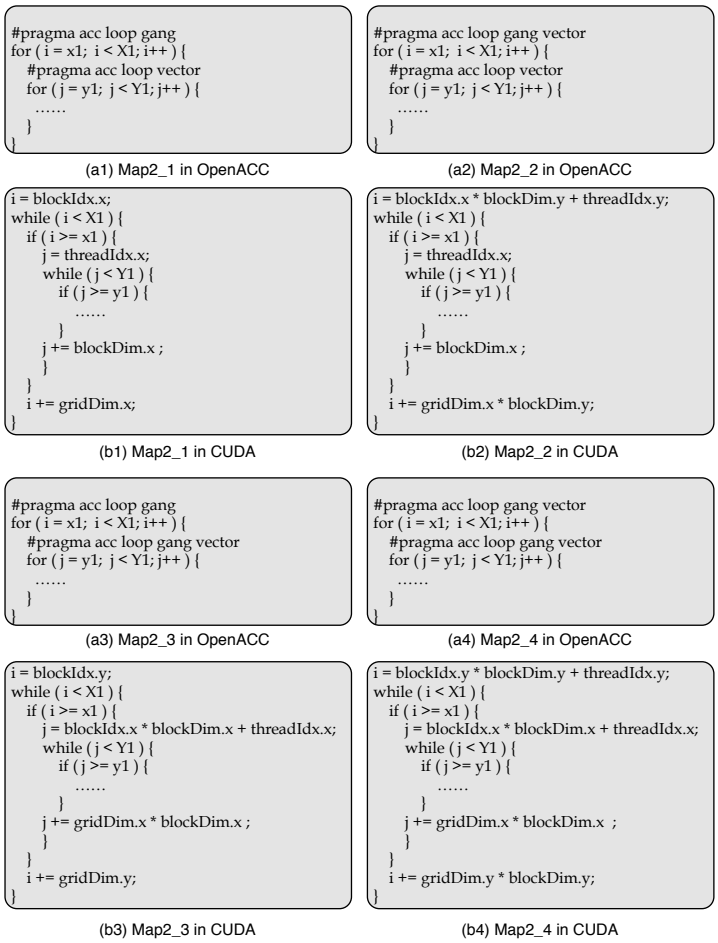


Fig. 6: Translated CUDA code from double nested loop mappings

- Two dimensional gang and three dimensional vector. After the mapping, the outmost loop stride in i, j, k axises are $blockDim.z$, $blockDim.y * griddim.y$ and $blockDim.x * griddim.x$. The translated CUDA code is shown as Map3_2.
- Both gang and vector are two dimensional. After the mapping, the outmost loop stride in i, j, k axises are $blockDim.x$, $blockDim.y * griddim.y$ and $gridDim.x * griddim.x$. The translated CUDA code is shown as Map3_3.

4 Runtime Support

The OpenACC annotated source code is parsed by the compiler to extract the device kernels and translate the OpenACC directives into runtime calls. Then

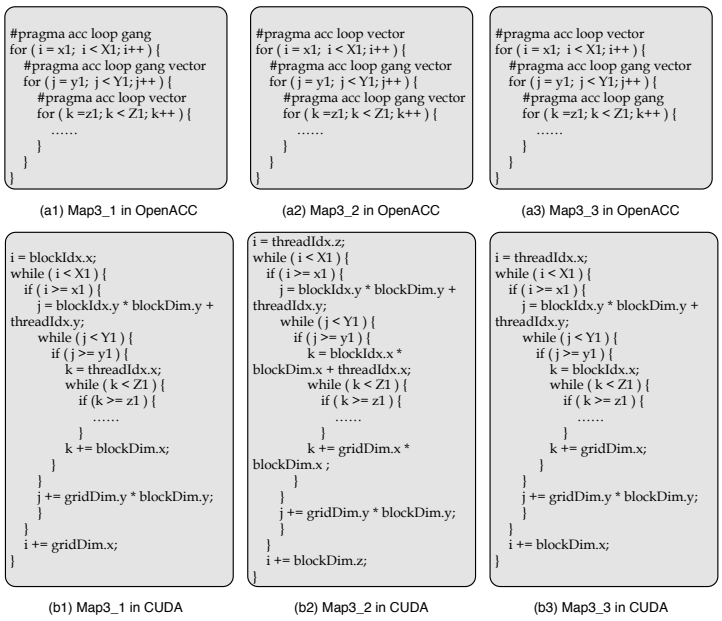


Fig. 7: Translated CUDA code from triple nested loop mappings

two parts of the code are generated: one part is the host code compiled by the host compiler, another part is the kernel code compiled by the accelerator compiler. The runtime is responsible for handling data movement and managing the execution of kernels from the host side.

4.1 Runtime Library Components

The runtime library consists of three modules: context module, memory manager, and kernel loader. The context module is in charge of creating and managing the virtual execution environment. This execution environment is maintained along the liftime of all OpenACC directives. All context and device related runtimes, such as `acc_init()` and `acc_shutdown()`, are managed by this module.

The memory manager helps to control the data movement between the host and device. The compiler will translate the clauses in `data` and `update` directives into corresponding runtime calls in this module. OpenACC provides a `present` clause that indicates the corresponding data list are already on the device, in order to avoid unnecessary data movement. To implement this feature, the runtime creates a global hash map that stores all the device data information. Whenever a compiler parses a `present` clause, it will translate this clause to the runtime call to check if the data list in the `present` clause are in the map. If the data exists in the map, then there is no need for data movement. If the data does not exist in the map, the compiler will issue a compilation error. Each data structure

in the map includes the host address, device address and the data size so that we can find the device address given a host address or vice versa. Note that the data allocated from `acc_malloc()` and the data in the `deviceptr` clause do not have a corresponding host address since they are only allowed to use on the device.

The purpose of kernel loader module is to launch the specified kernel from the host. After the kernel file is compiled by the accelerator compiler, the runtime loads the generated file, setups the threads topology and pushes the corresponding arguments list into the kernel parameter stack space, then launch the specified kernel. Since different kernels have different number of parameters, a vector data structure is created to store the kernel arguments to guarantee that the kernel argument size is dynamic. Another work to do before launching a kernel is to specify the threads topology. The compiler parses the loop mapping strategy and then generates the corresponding thread topology. The recommended threads value in the topology is described in section 4.2.

4.2 Gang and Vector Topology Setting

The threads topology is an important factor affecting application performance. Since we map gangs to blocks in grid and vector into threads within each block, the values of blocks and threads need to be chosen carefully. Too many blocks and threads may generate potential scheduling overhead, and too few threads and blocks cannot take advantage of the whole GPU hardware resources such as cache and registers. The threads topology setting should consider exposing enough parallelism in each multiprocessor and balancing the workload across all multiprocessors. Different threads topology affects the performance differently. Some results with different topology values are discussed in section 5. In OpenUH, if the user did not specify the gang and vector number, the default value will be used. The default vector size is 128 because the Kepler architecture has quad warp scheduler that allows to issue and execute four warps (32 threads) simultaneously. The default gang number is 16 since Kepler allows up to 16 thread blocks per multiprocessor.

4.3 Execution Flow in Runtime

Figure 8 gives a big picture of the execution flow at runtime. In the beginning, `acc_init()` is called to setup the execution context. This routine can be either called explicitly by the user or implicitly generated by the compiler. Next the data clauses will be processed. There are different kinds of data clauses (e.g. `copyin`, `copyout` and `copy`) and these data clauses may be in either of `data`, `parallel` or `kernels` directive. If the data needs to be accessed from the device, for instance those in `copyin` or `copy` or `update device` clauses, then they are transferred from the host to device. These data clauses will be scanned and processed. The purpose of this step is to make the data ready before launching the kernels. After the data is ready, we will setup the threads topology and push the corresponding arguments to the kernel. So far everything is ready and we can safely load and launch the kernel. If the kernel needs to do some reduction

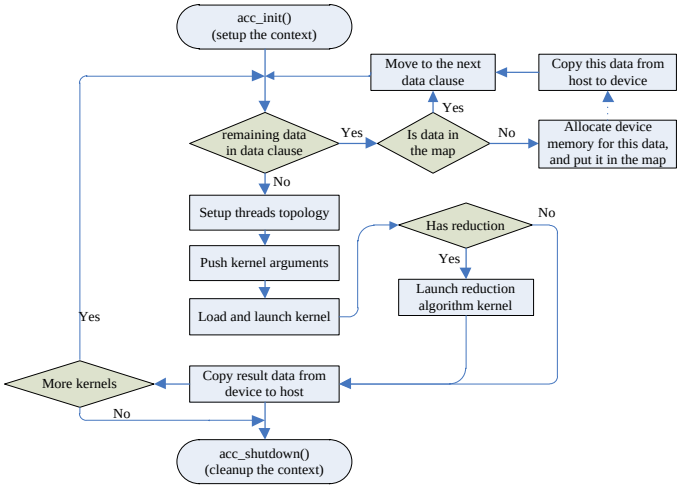


Fig. 8: Execution flow with OpenACC runtime library

operation, after this kernel is finished a separate reduction algorithm kernel will be launched. The result data, for instance those in `copyout` or `copy` or `update host` clauses, will be transferred from the device to host. Finally `acc_shutdown()` is called to release all the resources and destroy the context.

5 Preliminary Results

We evaluated OpenUH OpenACC compiler implementation using performance test suite from [3], Stencil benchmark from [13] and DGEMM written by ourselves. The double precision numerical algorithms in these examples are on either 2D or 3D grids, and therefore they are highly suitable to test different loop mapping strategies. The experimental machine has 16 cores Intel Xeon x86_64 CPU with 32GB main memory, and a NVIDIA Kepler GPU card (K20). OpenUH translates the original OpenACC program into host code and device code. The host code is compiled by gcc 4.4.7 with -O0 and the device code is compiled by nvcc 5.0 with "-arch=sm_35", and then they are linked into an executable.

5.1 Performance for Double Nested Loop Mapping

In the first stage, we compile these benchmarks with OpenUH compiler and compare the performance difference among different loop mappings. Figure 9 shows the performance comparison in different benchmarks with different double nested loop mappings. All of Jacobi, DGEMM and Gaussblur have double nested parallel loops but they show different performance behavior. In Jacobi,

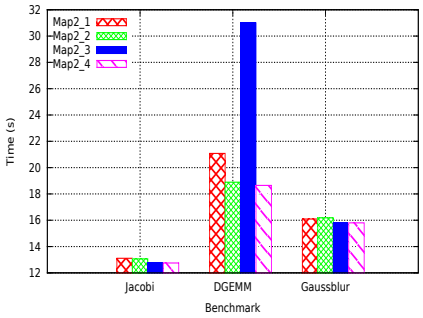


Fig. 9: Double nested loop mapping.

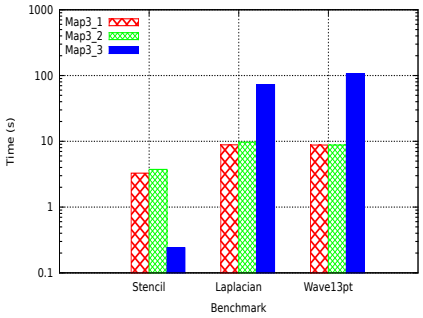


Fig. 10: Triple nested loop mapping.

Table 2: Threads used in each loop with double loop mappings

Benchmark	Double Loop	Map2_1	Map2_2	Map2_3	Map2_4
Jacobi (2048x2048)	Outer loop	2048	1024x2	2046	1023x2
	Inner loop	128	128	16x128	16x128
DGEMM (8192x8192)	Outer loop	8192	4096x2	8192	4096x2
	Inner loop	128	128	64x128	64x128
Gaussblur (1024x1024)	Outer loop	1024	512x2	1020	510x2
	Inner loop	128	128	8x128	8x128

the data accessed from the inner loop are contiguous in memory while they are non-contiguous when accessed from the outer loop. In all of our four double nested loop mappings, the inner loop uses `vector` which means the threads executing the inner loop are consecutive. In both `vector` and `gang vector` cases, the threads are consecutive and the only difference is the length of concurrent threads. In Jacobi inner loop, consecutive threads access aligned and consecutive data and therefore the memory access is coalesced. In this case the memory access pattern and the loop mapping mechanism match perfectly. That is why the performance using all of the four loop mappings are close. Table 2 shows the number of threads used in each loop mapping. Because Map2_1 and Map2_2 have less threads than Map2_3 and Map2_4 in the inner loop, the execution time is slightly longer. Map2_1 and Map2_2 have the same performance since their threads are the same in both the outer loop and inner loop. The performance behavior of Gaussblur is similar to Jacobi because their memory access pattern and threads management are similar.

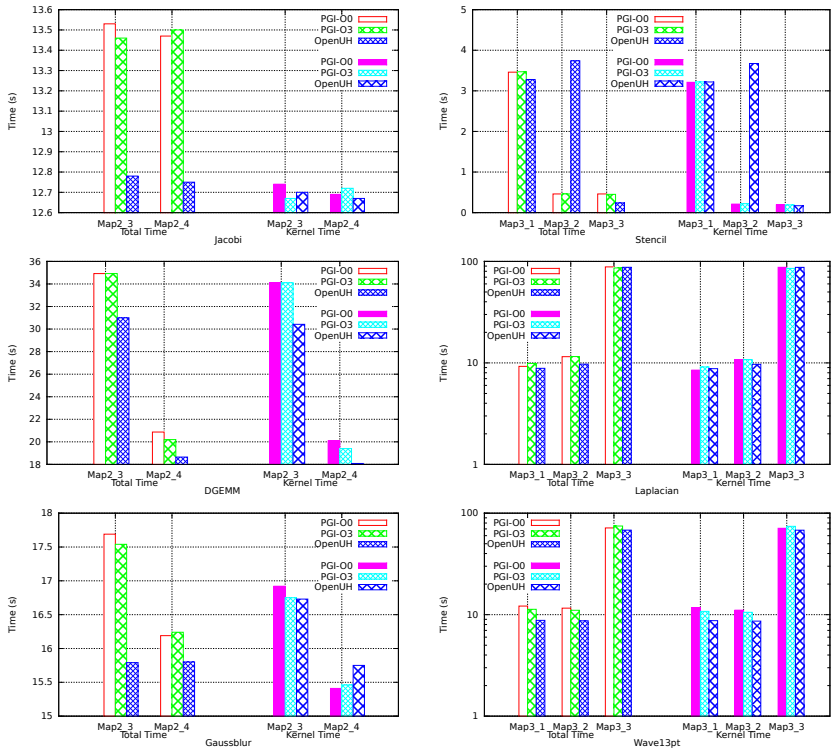
In DGEMM, the performance of Map2_2 and Map2_4 are better than the other two mappings which is because they both have enough parallelism in each block to hide memory access latency. The performance penalty in Map2_1 is due to less parallelism in each block. Map2_3 has the worst performance as it does not have enough parallelism in each block and has many thread blocks. Too many blocks means more scheduling overhead as a block cannot be started until all resources for a block is available.

5.2 Performance for Triple Nested Loop Mapping

Table 3: Threads used in each loop with triple loop mappings

Benchmark	Triple Loop	Map3_1	Map3_2	Map3_3
Stencil (512x512x64)	outermost loop	510	2	128
	middle loop	255x2	255x4	255x2
	innermost loop	128	16x64	62
Laplacian (128x128x128)	outermost loop	63	2	128
	middle loop	126x2	32x4	2
	innermost loop	128	2x64	126
Wave13pt (128x128x128)	outermost loop	64	2	128
	middle loop	124x2	31x4	2
	innermost loop	128	2x64	124

Figure 10 shows the performance comparison in different benchmarks with different triple nested loop mappings. In Stencil, the data is stored in memory in $x \rightarrow y \rightarrow z$ order which means the data is firstly stored in x dimension, then y dimension and lastly z dimension. The computation kernel, however, access the data in $z \rightarrow y \rightarrow x$ order which means the data accessed in the innermost loop (z dimension) are not contiguous in memory but the data accessed in the outermost loop (x dimension) are contiguous in memory. The loop Map3_3 uses `vector` in the outermost loop and therefore the global memory access are coalesced. This follows the most important rule when mapping the loop: consecutive threads access consecutive data in memory. Hence the performance with Map3_3 is much better than the other two loop mappings. Note that the loop Map3_2 also used `vector` in the first loop, but its performance is worse than Map3_3. This is because the threads in this `vector` are in z dimension and not consecutive in CUDA context. The loop Map3_1 uses `gang` in the first loop and this also indicates that the threads are not consecutive in this level, as the stride between each thread pair is *gridDim.x* rather than 1. Table 3 shows the threads in each loop of different benchmarks. In Stencil note that although the total number of threads in Map3_1 is much more than that of Map3_3, its performance is still poorer which is just because the memory access is uncoalesced. Laplacian and Wave13pt have similar performance patterns in which the performance with loop Map3_1 and 3_2 are much better than loop Map3_3. The reason is that their data layout in memory matches the data memory access pattern indicated by the loop mapping mechanism. For instance, in Laplacian the data accessed in the innter-most loop are consecutive in memory and the threads specified by loop Map3_1 and 3_2 are also consecutive, as a result the data accesses are coalesced in GPU and high performance can be achieved. With loop Map3_3, however, the used loop clause is `gang` and the stride between threads is larger than 1 which means the threads are not consecutive. As a consequence, the non-consecutive threads try to access consecutive data and therefore the data access is not coalesced, and finally the performance is penalized. The loop Map3_1 and 3_2 are similar and the only difference is the thread increment stride. That can be explained why the performance using these two loop mapping mechanisms are close.



(a) Double nested loop mapping (b) Triple nested loop mapping

Fig. 11: Performance comparison with different loop mappings

5.3 Performance Comparison between OpenUH and PGI OpenACC

We also compared the performance for all benchmarks with PGI commercial compiler. PGI 13.6 was used and both -O0 and -O3 optimization flags were experimented, respectively. OpenUH only used -O0 since it has not applied any optimization in generated GPU code. Figure 11 (a) shows the performance difference between OpenUH and PGI compiler in double nested loop mapping. Since PGI compiler always converts Map2_1 to 2_3 and Map2_2 to 2_4, we only compare the performance between 2_3 and 2_4 loop mappings. We measured the kernel time which indicates the efficiency of the kernel code generated by compiler, and the total time which includes the kernel time, data transfer time and the runtime overhead. The result shows that OpenUH is slightly better than PGI compiler in the total time of Jacobi, DGEMM and Gaussblur. By profiling all benchmarks, we found that the performance difference is due to PGI compiler always creates two contexts to manage asynchronous data transfer even though the `async` clause was not specified in the program. As a result, the runtime has more overhead of creating another context and managing the synchronization of

all asynchronous activities. For the kernel time, OpenUH is still slightly better than PGI in Jacobi and DGEMM, but slightly worse in Gaussblur. Overall the performance in PGI compiler with -O0 and -O3 has no much difference, and the performance variance between OpenUH and PGI is within a very small range and OpenUH performance is very competitive comparing to PGI compiler.

Figure 11 (b) shows the performance comparison between OpenUH and PGI compiler in triple nested loop mappings. It is observed that in Stencil the performance of OpenUH is much worse than PGI compiler in loop Map3_2. We believe that PGI did some memory access pattern analysis and can automatically adjust its loop mapping mechanism, thus delivering better performance than ours. Briefly speaking, in the outermost loop of Stencil, the data access is not coalesced in OpenUH implementation as OpenUH assumes the data accessed only from the innermost loop are contiguous in memory, whereas in this program the data accessed only from the outermost loop are contiguous in memory. We believe PGI compiler did data flow analysis which can automatically detect this and change the loop mapping, so that the access to the outermost loop are coalesced by threads. So far OpenUH has implemented the same loop mapping techniques, but it requires a memory access analysis model to dynamically change the loop mapping, which is one of our ongoing work.

6 Related Work

There are both commercial OpenACC compiler and academic compiler efforts to support high-level programming models for GPGPUs. CAPS compiler [1] also uses the same source-to-source translation approach as ours. PGI OpenACC accelerator compiler [4] use binary code generation approach. Cray compiler [7] is another OpenACC compiler that can only be used in Cray supercomputers. These three compilers have different mapping mechanisms as we discussed in early section. Since both CAPS and Cray have different interpretations of `gang`, `worker` and `vector`, we did not compare our results with these compilers for fairness reason. accULL [12] is another OpenACC compiler written in python script. KernelGen [11] can port the existing code into Nvidia GPU without the need of adding any directives. It requires the GPU to support dynamic parallelism, so it is not as portable as OpenACC. OpenMPC [9] translates OpenMP code to cuda and HiCUDA [8] is another directive-based model which is simlar to OpenACC but the user still needs to manage almost everything.

7 Conclusion

In this paper, we presented our effort of creating an OpenACC compiler in our OpenUH compiler framework. We have designed loop mapping mechamisms of single, double nested, and triple nested loops that are used in the compiler transformation. These mechanisms will be helpful for users to adopt suitable computation distribution techniques according to their application's characteristics. Our open-source OpenUH compiler can generate readable CPU code and

CUDA code, which allows user to further tune the code performance. The experiments show that our compiler can generate code with competitive performance to commercial OpenACC compiler.

Since this is our first baseline version, advanced features such as multi-dimensional array movement, loop collapse, parallel construct and async, etc. are under development. All the executions currently implemented are on synchronization mode. Advanced compiler analysis and transformation techniques will also be explored to further improve the quality of generated kernel codes.

8 Acknowledgements

This work was supported in part by the NVIDIA and Department of Energy under Award Agreement No. DE-FC02-12ER26099. We would also like to thank PGI for providing the compilers and support for the evaluation.

References

1. CAPS Enterprise OpenACC Compiler Reference Manual. `http://www.openacc.org/sites/default/files/HMPPOpenACC-3.2_ReferenceManual.pdf`, June 2013.
2. CAPS OpenACC Parallism Mapping. `http://kb.caps-entreprise.com/what-gang-workers-and-threads-correspond-to-on-a-cuda-card`, 2013.
3. Performance Test Suite. `https://hpcforge.org/plugins/mediawiki/wiki/kernelgen/index.php/Performance_Test_Suite`, June 2013.
4. PGI Compilers. `http://www.pgroup.com/resources/accel.htm`, June 2013.
5. The OpenACC Standard. `http://www.openacc-standard.org`, June 2013.
6. M. W. Brent Leback and D. miles. The PGI Fortran and C99 OpenACC Compilers. *Cray User Group*, 2012.
7. C. Cray. C++ Reference Manual, 2003.
8. T. D. Han and T. S. Abdelrahman. hiCUDA: High-Level GPGPU Programming. *IEEE Transactions on Parallel and Distributed Systems*, 22:78–90, 2011.
9. S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.
10. C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. OpenUH: An Optimizing, Portable OpenMP Compiler. *Concurrency and Computation: Practice and Experience*, 19(18):2317–2332, 2007.
11. D. Mikushin and N. Likhogrud. KERNELGEN - A Toolchain for Automatic GPU-centric Applications Porting. `https://hpcforge.org/scm/viewvc.php/*checkout*/doc/sc_2012/sc_2012.pdf?root=kernelgen`, 2012.
12. R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. de Sande. accULL: An OpenACC Implementation with CUDA and OpenCL Support. In *Euro-Par 2012 Parallel Processing*, pages 871–882. Springer, 2012.
13. J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. *Center for Reliable and High-Performance Computing*, 2012.