NAS Parallel Benchmarks for GPGPUs using a Directive-based Programming Model

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

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

Abstract. The broad adoption of accelerators boosts the interest in accelerator programming. Accelerators such as GPGPUs are optimized for throughput and offer high GFLOPS and memory bandwidth. CUDA has been adopted quite rapidly but it is proprietary and only applicable to GPUs, and the difficulty in writing efficient CUDA code has kindled the necessity to create higher-level programming approaches such as OpenACC. Directive-based programming models such as OpenMP and OpenACC offer programmers an option to rapidly create prototype applications by adding annotations to guide compiler optimizations. In this paper we study the effectiveness of a high-level directive based programming model, OpenACC, for parallelizing NAS Parallel Benchmarks (NPB) on GPGPUs. We present the application of techniques such as array privatization, memory coalescing, cache optimization and examine their impact on the performance of the benchmarks. The right choice or combination of techniques/hints are crucial for compilers to generate highly efficient codes tuned to a particular type of accelerator. Poorly selected choice or combination of techniques can lead to degraded performance. We also propose a new clause, 'scan', that handles scan operations for arbitrary input array size. We hope that the practices discussed in this paper will provide useful guidance to users to effectively migrate their sequential/CPU-parallel codes to GPGPU architectures and achieve optimal performance.

1 Introduction

Heterogeneous architectures that comprise of commodity CPU processors and computational accelerators such as GPGPUs have been increasingly adopted in both supercomputers and workstations/desktops for engineering and scientific computing. These architectures are able to provide massively parallel computing capabilities provided by accelerators while preserving the flexibilities of CPU accommodating computation of different workloads. However, effectively tapping their full potential is not straight-forward, largely due to the programmability challenges faced by users while mapping highly computation algorithms.

Programming models such as CUDA[7] and OpenCL[4] for GPGPUs offer users programming interfaces with execution models closely matching the GPU

architectures. Effectively using these interfaces for creating highly optimized applications require programmers to thoroughly understand the underlying architecture, as well as significantly change the program structures and algorithms. This affects both productivity and performance. Another approach that has been standardized are the high-level directive-based programming models, e.g. HMPP[14], OpenACC [3] and OpenMP [5]. These models require developers to insert directives and runtime calls into the existing source code, offloading portions of Fortran or C/C++ codes to be executed on accelerators.

Directives are high-level language constructs that programmers can use to provide useful hints to compilers to perform certain transformation and optimizations on the annotated code region. The use of directives can significantly improve programming productivity. Users can still achieve high performance of their program comparable to code written in CUDA or OpenCL, subjected to the requirements that a careful choice of directives and compiler optimization strategies are made. The choice of such strategies vary from one accelerator type to the other.

In this paper, we will discuss the parallelization strategies to port NAS Parallel Benchmarks (NPB) [10] to GPGPUs using high-level compiler directives, OpenACC. NPB are well recognized for evaluating current and emerging multi-core/many-core hardware architectures, characterizing parallel programming models and testing compiler implementations. The suite consists of five parallel kernels (IS, EP, CG, MG and FT) and three simulated computational fluid dynamics (CFD) applications (LU, SP and BT) derived from important classes of aerophysics applications. Together they mimic the computation and data movement characteristics of large scale computational CFD applications [10]. This is one of the standard benchmarks that is close to real world applications. We believe that the OpenACC programming techniques used in this paper can be applicable to other models such as OpenMP. Based on the application requirements, we will analyze the applicability of optimization strategies such as array privatization, memory coalescing and cache optimization. With vigorous experimental analysis, we will then analyze how the performance can be incrementally tuned.

The main contributions of this paper are:

- With GPUs as the target architecture, we highlight the critical techniques and practices required to parallelize benchmarks in NAS that are close to real-world applications.
- We analyze a number of choices and combinations of optimization techniques and study their impact on application performance. We learned that poorly selected options or using system default options for optimizations may lead to significant performance degradation. We share these findings in this paper.
- We also compare the performances of OpenACC NPB with that of the well-tuned OpenCL and CUDA versions of the benchmarks to present the reasoning behind the performance gap.

To the best of our knowledge, we are the first group to create an OpenACC benchmark suite for the C programs in NPB.

The organization of this paper is as follows: Section 2 provides an overview of GPU architecture and OpenACC. In section 3 we discuss typical steps of parallelizing scientific application using OpenACC and some optimization techniques. Performance results are discussed in Section 4. Section 5 discusses the programmability and performance portability issues of using OpenACC. Section 6 highlights related work in this area. We conclude our work in Section 7.

2 GPU Architecture and OpenACC Directives

GPU architectures differ significantly from that of traditional processors. In this paper, GPU is always referred to **G**eneral **P**urpose **GPU**. Employing a Single Instruction Multiple Threads (SIMT) architecture, NVIDIA GPUs have hundreds of cores that can process thousands of software threads simultaneously. GPUs organize both hardware cores and software threads into two-level of parallelism. Hardware cores are organized into an array of SMs (Streaming Multiprocessors), each SM consisting of a number of core named as SPs (Scalar Processors). An execution of a computational kernel, e.g. CUDA kernel, will launch a set of (software) thread blocks. Each thread block can contain hundreds of threads. For programmers, the challenges to efficiently utilize the massive parallel capabilities of GPUs are to map the algorithms onto two-level thread hierarchy, and to lay out data on both the global memory and shared memory to maximize coalesced memory access for the threads. Using low-level programming models such as CUDA and OpenCL to do this has been known as not only time consuming but also the software created are not identical to its original algorithms significantly decreasing code readability.

Directive-based high-level programming models for accelerators, e.g. OpenACC and OpenMP accelerator extensions, have been created to address this programmability challenge of GPUs. Using these programming models, programmers insert compiler directives into a program to annotate portions of code to be offloaded onto accelerators for executions. This approach relies heavily on the compiler to generate efficient code for thread mapping and data layout. It could be potentially challenging to extract optimal performance using such an approach rather than using other explicit programming models. However, the model simplifies programming on heterogeneous systems thus saving development time, while also preserves the original code structure that helps in code portability. These models extend the host-centric parallel execution model for devices that reside in separate memory spaces. The execution model assumes that the main program runs on the host, while the compute-intensive regions of the main program are offloaded to the attached accelerator. The memory spaces on the host and the device are separate from one another. Host cannot access the device memory directly and vice versa.

OpenACC allows users to specify three levels of parallelism in a data parallel region: coarse-grain parallelism “gang”, fine-grain parallelism “worker” and vector parallelism “vector”, to map to the multiple-level thread hierarchy of GPUs. Mapping these three-level parallelism to the GPU threading structure will be left

to the compiler and runtime systems, according to the hints given by the programmers. It can be a challenge for programmers, particularly on large programs with complex irregular data access pattern and thread synchronization.

There are already a number of compilers that provides support for OpenACC. Those include PGI, CAPS and Cray, open-source OpenACC compilers include accULL [22], OpenUH [24] and OpenARC [19]. Our focus in this paper is to use the NAS benchmarks to evaluate OpenACC support in our in-house OpenUH compiler.

3 Parallelization and Optimization Strategies

One of the main benefits of programming using a directive-based programming model is achieving performance by adding directives incrementally and creating portable modifications to an existing code. We consider the OpenMP version of NPB benchmarks as the starting point. Steps to parallelize legacy code using OpenACC are:

1) Profile the application to find the compute intensive parts, which are usually loops.

2) Determine whether the compute intensive loops can be executed in parallel. If not, perform necessary code transformations to make the loops parallelizable, if possible.

3) Prepend `parallel/kernels` directives to these loops. The `kernels` directive indicates that the loop needs to be executed on the accelerator. Using the `parallel` directive alone will cause the threads to run the annotated code block redundantly, until a `loop` directive is encountered. The `parallel` directive is mostly effective for non-loop statements.

4) Add `data` directives for data movement between the host and the device. This directive should be used with care to avoid redundant data movement, e.g. putting `data` directives across multiple compute regions. Inside the data region, if the host or device needs some data at the end of one compute region, `update host` directive could be used to synchronize the corresponding data from the device to host, or `update device` directive is used if the device needs some data from the host.

5) Optimize data structures and array access pattern to efficiently use the device memory. For instance, accessing data in the global memory in a coalesced way, i.e. consecutive threads should access consecutive memory address. This may require some loop optimizations like loop permutation, or transforming the data layout that will change the memory access pattern.

6) Apply loop scheduling tuning. Most of the OpenACC compilers provide some feedback during compilation informing users about how a loop is scheduled. If the user finds the default loop scheduling not optimal, the user should optimize the loop manually by adding more `loop` directives. This should lead to improvement in speedup.

7) Use other advanced optimizations such as the `cache` directive, which defines the variables to be cached by the kernel. Usage of the `async` clause will

```
for(k=0; k<N; k++){
    for(j=0; j<N; j++){
        for(i=0; i<N; i++){
            A[j][i] = ...
        }
    }
}
```

```
for(k=0; k<N; k++){
    for(j=0; j<N; j++){
        for(i=0; i<N; i++){
            AX[k][j][i] = ...
        }
    }
}
```

Fig. 1: Array privatization example

initiate data movement operations and kernel execution as asynchronous activities, thus enabling an overlap with continuous execution by the host CPU.

Some of the above steps need to be applied repeatedly along with profiling and feedback information provided by compiler and profilers. The practices and optimization techniques applied vary depending on the original parallel pattern and code structures of an application. Some of those techniques are summarized in the following sections. While these techniques have been used for optimizing parallel program on CPUs, applying them on GPUs pose different challenges, particularly when using them in large code bases.

3.1 Array Privatization

Array privatization makes different threads access distinct memory addresses, so that different threads do not access the same memory address. It is a technique of taking some data that is common or shared among parallel tasks and duplicating it so that different parallel tasks can have a private copy to operate. Figure 1 shows an example for array privatization. If we parallelize the triple-nested loop on the left side of the figure using OpenMP for CPU and only parallelize the outermost loop, each thread handles the inner two loops. The array A could be annotated as OpenMP `private` clause to each thread, thus no modification is required to keep the memory usage minimal and improve the cache performance. However this is not the case with OpenACC. In OpenACC, if the compiler still only parallelizes the outermost loop, multiple threads will be reading and writing to the same elements of the array A. This will cause data race conditions, incorrect results and potential crashes. An option here is to use the OpenACC `private` clause which is described in [6]. However, if the number of threads is very large, as typically in GPUs, it is very easy that all copies of the array exceed the total memory available. Even though sometimes the required memory does not exceed the available device memory, it is possible that the assigned number of threads is larger than the number of loop iterations, and in this case some of the device memory will be wasted since some threads are idle. Also the life time of variable within a `private` clause is only for a single kernel instance. This limits our choice to apply loop scheduling techniques since only the outermost loop can be parallelized. If the triple nested loop can be parallelized and each thread executes the innermost statements, thousands of threads still need to be created. Keeping the array A private to each thread will easily cause an overflow

```
     #pragma acc kernels loop gang
80:  for (k = 0; k <= grid_points[2]-1; k++) {
       #pragma acc loop worker
81:    for (j = 0; j <= grid_points[1]-1; j++) {
         #pragma acc loop vector
82:      for (i = 0; i <= grid_points[0]-1; i++) {
83:        for (m = 0; m < 5; m++) {
84:            rhs[m][k][j][i] = forcing[m][k][j][i];
85:         }
86:      }
87:    }
88:  }
```

Fig. 2: Loop scheduling example

of memory available on the accelerator device. The right side of Figure 1 shows the array privatized code that addresses this issue. This solution added another dimension to the original array so that all threads can access different memory addresses of the data and no data race will happen anymore.

3.2 Loop Scheduling Tuning

When parallelizing loops using OpenACC, `parallel/kernels` directives are inserted around the loop region. With the `parallel` directive, the user can explicitly specify how the loop is scheduled by setting whether the loop is scheduled in the level of `gang`, `worker` or `vector`. With the `kernels` directive, however, loop scheduling is usually left to the compiler's discretion. Ideally, the compiler performs loop analysis and determines an optimal loop scheduling strategy. Our simple experiments show that, when using the `kernels` directive, the compiler makes good choices most of the times. But the compiler often opts for the less efficient loop scheduling when the loop level is more than three. Figure 2 shows one of the scheduling techniques that delivers efficient loop scheduling. However the default scheduling by some compiler only applies to the loops in lines 82 and 83. The loops in line 80 and 81 are executed sequentially. This default option is very inefficient since the two outer most loops are not parallelized. Work in [24] discusses other loop scheduling mechanisms that could be applied in this context.

3.3 Memory Coalescing Optimization

The speedup from the parallel processing capability of GPU can be tremendous if memory coalescing is efficiently achieved. GPU has faster memory with unique data fetching and locality mechanism. In CPU, only one thread fetches consecutive memory data into the cache line, so the data locality is limited to only one thread. In GPU, however, consecutive threads fetch consecutive memory data into the cache line allowing better data locality. For instance, the code in Figure 2 is already optimized for memory coalescing. The i loop is vectorized with the rightmost dimension of *rhs* and *forcing* is i. In the original serial

code version, the memory access pattern of *rhs* and *forcing* are rhs[k][j][i][m] and forcing[k][j][i][m], respectively. But for memory coalescing purposes, we need to reorganize the data layout so that the dimension "m" is not on the farther right. Since C language is row-major, the right most dimension is contiguous in memory. We need the threads to access (i.e. the vector loop) the right most dimension. So after data layout reorganization, the memory access pattern becomes rhs[m][k][j][i] and forcing[m][k][j][i].

3.4 Data Motion Optimization

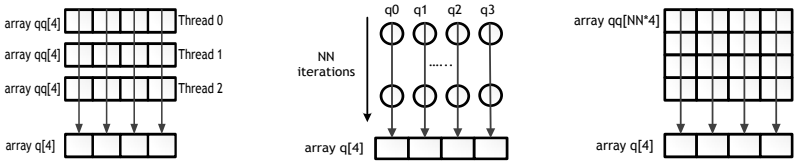
Data transfer overhead is one of the important factors to consider when determining whether it is worthwhile to accelerate a workload on accelerators. Most of the NPB benchmarks consist of many global variables that persist throughout the entire program. An option to reduce data transfer will be to allocate the memory for those global variables at the beginning of the program so that those data reside on the device until the end of the program. Since some portion of the code cannot be ported to the device, we could use `update` directive to synchronize the data between the host and device.

3.5 Cache Optimization

NVIDIA Kepler GPU memory hierarchy has several levels of memory, including global memory, then L2 cache for all SMs and the registers, L1 cache, shared memory and read-only data cache for each SM. In Kepler GPU, L1 cache is reserved only for local memory accesses such as register spilling and stack data. Global loads are cached in L2 cache only [8]. Here the usage of both L1 and L2 is controlled by the hardware and they are not manageable by the programmer. The shared memory can be utilized by the `cache` directive in OpenACC. Although the read-only data cache is also controlled by the hardware, the programmer can give some hints in the CUDA kernel file to tell the compiler what the read-only data list is. Since the read-only data cache is a device specific memory, OpenACC does not have any directive to utilize this cache. However, when the user specifies the device type when using OpenACC, the compiler can perform some optimizations specific to the specified device. We implemented this optimization in the compiler used so that the compiler can automatically determine the read-only data in a kernel by scanning all data in that kernel and then add "const __restrict__" for all read-only data and add "__restrict__" for other data that has no pointer alias issue. These prefix are required in CUDA if the user wants the hardware to cache the read-only data [8]. This compiler optimization can improve the performance significantly if the read-only data is heavily reused.

3.6 Array Reduction Optimization

Array reduction means every element of an array needs to do reduction. This is supported in OpenACC specification which only supports scalar reduction.



(a) OpenMP solution (b) OpenACC solution 1 (c) OpenACC solution 2

Fig. 3: Solutions of array reduction in EP benchmark.

Different programming models solve this issue differently. As shown in Figure 3, there are several ways to solve the array reduction in array q in EP benchmark. In the OpenMP version, each thread has its own private array qq to store the partial count of q, for the purpose of reducing the overhead of atomic update of shared variables. Thus, each thread only needs to perform an atomic update on q with its own partial sum qq. Since OpenACC does not support array reduction, Lee et al. [20] decomposes the array reduction into a set of scalar reductions which is shown in Figure 3 (b). This implementation is not scalable as it cannot handle large array reduction, and the size of the result array must be known at compile time. Our solution, as shown in Figure 3 (c), uses the array privatization technique to make a copy of q and expand it by another dimension with size NN (declared as new variable qq). In this way, each thread does its own work independently and writes the result into its own portion of the global memory. Finally, each element of q can be obtained by doing reduction just once with qq.

3.7 Scan Operation Optimization

The NAS IS benchmark has both inclusive and exclusive prefix-sum/scan operations. The inclusive scan takes a binary operator $\bigoplus$ and an array of N elements $[a_0, a_1, ..., a_{N-1}]$ and returns the array $[a_0, (a_0 \bigoplus a_1), ..., (a_0 \bigoplus a_1 \bigoplus ... \bigoplus a_{N-1})]$. Exclusive scan is defined similarly but shifts the output and uses an identity value I as the first element. The output array is $[\mathrm{I}, a_0, (a_0 \bigoplus a_1), ..., (a_0 \bigoplus a_1 \bigoplus ... \bigoplus a_{N-2})]$. In scan loop, an element in the output array depends on its previous element, and because of such data dependence, it cannot be parallelized by the `loop` directive in OpenACC. To overcome such limitations, we provided some extensions to the OpenACC standard. We introduced a new `scan` clause to the `loop` directive followed by usage of recursive algorithm in [17] to handle the scan operation for arbitrary input array size. We implemented this optimization in OpenUH compiler.

4 Performance Evaluation

The experimental setup is a machine with 16 cores Intel Xeon x86_64 CPU with 32GB main memory, and an NVIDIA Kepler GPU card (K20) with 5GB global

Table 1: Comparing elapsed time for NPB-ACC, NPB-SER, NPB-OCL and NPB-CUDA (time in seconds), "-" implies no result due to "out ofmemory" issue. For NPB-CUDA, only LU, BT and SP are accessible. Data size increases from A to C, ∼16x size increase from each of the previous classes. The "Techniques Applied" numbers refer to the optimizations described in corresponding sections. Other than listed techniques, we have optimized all of our OpenACC implementations including using data motion optimizations as well.

Benchmark	EP			CG			FT			IS		
Data Size	A	B	C	A	B	C	A	B	C	A	B	C
NPB-SER	46.56	187.02	752.03	2.04	101.80	269.96	6.97	79.42	390.35	0.99	4.04	17.00
NPB-OCL	0.27	0.82	2.73	0.36	13.42	35.39	1.49	32.55	-	0.04	0.35	1.74
NPB-ACC	0.49	1.96	7.85	0.36	9.51	21.28	1.18	9.20	-	0.06	0.23	1.94
Techniques Applied	3.1, 3.3, 3.6			3.5			3.1, 3.3			3.7		
Benchmark	MG			LU			BT			SP		
Data Size	A	B	C	A	B	C	A	B	C	A	B	C
NPB-SER	2.57	11.48	99.39	60.38	264.71	1178.97	93.14	387.51	1626.33	52.17	225.26	929.85
NPB-OCL	0.13	0.61	5.48	5.32	16.70	54.88	46.12	167.48	-	11.84	54.35	288.40
NPB-ACC	0.24	1.12	7.55	6.64	26.12	103.97	15.25	63.61	226.70	3.45	15.90	57.46
NPB-CUDA	-	-	-	5.79	19.58	75.06	13.08	53.46	216.98	2.47	11.17	43.16
Techniques Applied	3.1, 3.2, 3.5			3,1, 3.3, 3.5			3,1, 3,2, 3.3, 3.5			3,1, 3.2, 3.3		

memory. We use OpenUH compiler to evaluate the performance of C programs of NPB on GPUs. This open source compiler provides support for OpenACC 1.0 at the time of writing this paper. Although implementations for OpenACC 2.0 are beginning to exist, they are not robust enough to be used to evaluate NPB-type benchmarks. For evaluation purposes, we compare the performances of our OpenACC programs with serial and third-party well tuned OpenCL [23] and CUDA programs [1] (that we had access to) of the NAS benchmarks. All OpenCL benchmarks run on GPU rather than CPU. We used GCC 4.4.7 and -O3 flag for optimization purposes. The CUDA version used by the OpenACC compiler is CUDA 5.5. The OpenCL codes are compiled by GCC compiler and link to CUDA OpenCL library.

Table 1 shows the execution time taken by NPB-SER, NPB-OCL and NPB-ACC, which are the serial, OpenCL and OpenACC versions of the NPB benchmarks, respectively. For the FT benchmark, OpenCL and OpenACC could not execute for problem size Class C. The reason being, FT is memory limited; the Kepler card in use ran out of memory. Same to do with the OpenCL program for BT benchmark. However this was not the case with OpenACC. The reason being: OpenCL allocated the device memory for all the data needed in the beginning of the application. With OpenACC program, different solver routines have different memory coalescing requirements, as a result, different routines have different data layout. For those data, OpenACC program only allocates the device memory in the beginning of the solver routines and frees the device memory before exiting these routines. This explains that the data in the OpenCL pro-

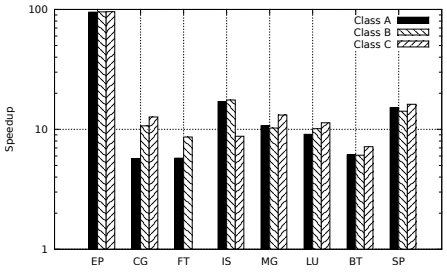


Fig. 4: NPB-ACC speedup over NPB-SER

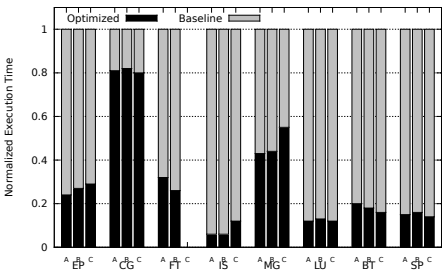


Fig. 5: NPB-ACC Performance improvement after optimization

gram are active throughout the full application, but for the OpenACC program, some data is only active in some of the routines, hence saving the total memory requirement at a given time.

Figure 4 shows the speedup of NPB-ACC over NPB-SER for the benchmarks that have been optimized. It is quite clear that all the benchmarks show significant speedup, especially EP. This is because EP is an embarrasingly parallel benchmark that has only few data transfers and our optimization technique enabled all the memory accesses to be nicely coalesced. Most of the benchmarks observed increase in speedup as the data problem size increased, except IS. This is because, IS uses buckets to sort an input integer array, the value of the bucket size is fixed as defined in the benchmark, no matter what the data size is. As a result, when the data size becomes larger, the contention to each bucket becomes more intense decreasing the performance to quite an extent. However this does not affect the numerical correctness due to atomic operations in place to prevent data races.

We measure the effectiveness of the potential optimizations applied in Figure 5 by comparing the baseline and the optimized versions of the benchmarks. The baseline versions use only array privatization in order to parallelize the code and data motion optimization to eliminate unnecessary data transfer overhead and not any other optimizations discussed. The optimized versions exploit the optimizations discussed earlier.

IS benchmark demonstrates much improvement from the baseline version. This is due to the *scan* operation discussed earlier. CG mainly benefits from cache optimization, the rest of the optimizations all seem to have a major impact on the benchmark's performance. FT benchmark shows improvement due to Array of Structure (AoS) to Structure of Array (SoA) transformation since the memory is not coalesced in AoS data layout but coalesced in SoA data layout. Note that the execution time of the three pseudo application benchmarks LU, BT and SP are even less than 20% of the time taken by the baseline version. LU and BT observed over ∼50% and ∼13% of performance improvement using cache optimization, since both the benchmarks extensively use read-only data.

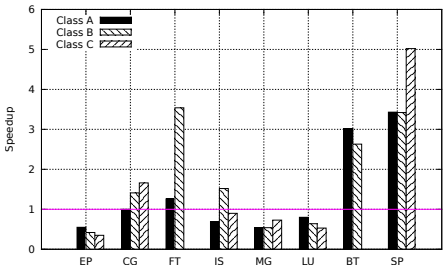


Fig. 6: NPB-ACC speedup over NPB-OCL

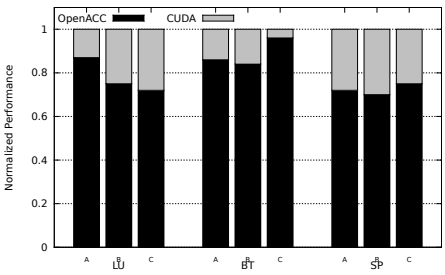


Fig. 7: NPB-ACC performance comparison with NPB-CUDA

LU, BT and SP benchmarks benefit significantly from memory coalescing optimizations since in the serial code the memory access is not coalesced at all for GPU architecture. Memory coalescing requires explicit data layout transformation. We observed that tuning loop scheduling is very crucial for MG, BT and SP benchmarks since these benchmarks have three or more levels of nested loops. The compiler could not always identify the best loop scheduling option, requiring the user to intervene.

These analysis of benchmark results indicate that it is insufficient to simply insert directives to an application no matter how simple or complex it is. It is highly essential to explore optimization techniques, several of those discussed in this paper, to not only give the compiler adequate hints to perform the necessary transformations but also perform transformations that can exploit the target hardware efficiently.

To evaluate our optimizations further, we compare the NPB-ACC with well-tuned code written with the low-level languages OpenCL (NPB-OCL) and CUDA (NPB-CUDA). Figure 6 and Figure 7 show the corresponding results. Figure 6 shows that the EP program using OpenACC is around 50% slower than that of the OpenCL program. This is because the OpenACC version uses array privatization, which increases the device memory in turn exceeding the available memory limit. Therefore we use the blocking algorithm to move data chunk by chunk into the device. We launch the same kernel multiple times to process each data chunk. The OpenCL program, however, uses the shared memory in GPU and does not need to use array privatization to increase the GPU device memory, therefore it only needs to launch the kernel once. Faster memory access through shared memory and reduced overhead due to less number of kernel launches improved the results for OpenCL. Although OpenACC provides a `cache` directive that has similar functionalities to CUDA's shared memory, the implementation of this directive within OpenACC compiler is not technically mature enough yet. This is one of the potential areas where support in OpenACC can be improved.

Performance of OpenACC programs of benchmarks BT and SP are much better than that of the OpenCL programs. The reason is two-fold. First up, the OpenCL program does not apply the memory coalescing optimization; mem-

ory accesses are highly uncoalesced. Secondly, the program does not apply loop fission optimization; there are very large kernels. Although the large kernel contains many parallelizable loops, they are only executed sequentially inside the large kernel. On the contrary, the OpenACC program uses loop fission, thus breaking the large kernel into multiple smaller kernels and therefore exposing more parallelism.

The OpenACC program for benchmark MG appears to be slower than that of the OpenCL program. This is because former program uses array privatization, which needs to allocate the device memory dynamically in some routines, however the latter uses shared memory, which has faster memory access and no memory allocation overhead. The OpenACC program for benchmark FT is faster than OpenCL, since OpenACC transforms the AoS to SoA data layout to enable memory coalescing. The OpenACC program for benchmark LU is slower than OpenCL since the former privatizes small arrays into the GPU global memory, but OpenCL uses the small array inside the kernel as in they will be allocated in registers or possibly spilled to L1 cache. The memory access from either register or L1 cache is much faster than that from the global memory as used by OpenACC.

Figure 7 shows the normalized performance of NPB-ACC and NPB-CUDA. We found CUDA programs for only the pseudo applications, i.e. LU, BT and SP, hence we have only compared OpenACC results of these applications with CUDA. The result shows that OpenACC programs for LU, BT and SP benchmarks achieve 72%∼87%, 86%∼96% and 72%∼75% to that of the CUDA programs, respectively. The range denotes results for problem sizes from CLASS A to C. We see that the performance gap between CUDA and OpenCL is quite small. The reasoning for the small performance gap is the same as that we have explained for the OpenCL LU benchmark. It is quite evident that careful choice of optimization techniques for high-level programming models can result in reaching performance very close to that of a well hand-written CUDA code. We believe that as the OpenACC standard and its implementation evolve, we might even be able to obtain better performance than CUDA. Thus successfully achieving portability as well.

5 Discussion

5.1 Programmability

Programming heterogeneous systems can be simplified using OpenACC-like directive-based approaches. An expected advantage is that they help maintain a single code base catering to multiple targets, leading to considerably lesser code maintenance. However, in order to achieve good performance, it is insufficient to simply insert annotations. The user's intervention is required to manually apply certain code transformations. This is because the compiler is not intelligent enough to determine the optimal loop scheduling for accelerated kernels and optimize the data movement automatically. With respect to memory coalescing requirement, currently there is no efficient mechanism to maintain different data

layout for different devices, the user has to change the data layout. There is no compiler support that can effectively utilize the registers and shared memory in GPU that play an important role in GPUs. Data movement is one of the most important optimization techniques. So far it has been the user's responsibility to choose the necessary data clause and to move data around in order to get the best performance. If the compiler provides suitable hints, this technique can prove to be quite useful.

5.2 Performance Portability

Achieving performance portability can be quite tricky. Different architectures demand different programming requirements. Merely considering a CPU and a GPU; obtaining optimal performance from CPU largely depends on locality of references. This holds good for GPUs as well, but the locality mechanism of the two architectures are different. The amount of computation that a CPU and a GPU can handle also differs significantly. It is not possible to maintain a single code base for two different architectures unless the compiler automatically handles most of the optimizations internally. Performance portability is not only an issue with just the architecture, but also an issue that different compilers can provide a different implementation for a directive/clause. Moreover the quality of the compilation matters significantly. For example, the OpenACC standard allows the user to use either `parallel` or `kernels` in the compute region. The `kernels` directive allows the compiler to choose the loop scheduling technique to be applied i.e. analyze and schedule each loop level to `gang/worker/vector`. A compiler can use its own technique to schedule the loop nest to nested gang, worker and vector; this is typically not part of the programming model standard. As a result, the performance obtained using the `kernels` directive is different for different compilers. On the contrary, the code that uses `parallel loop` directive is more portable since this allows the user to have control over adopting the loop scheduling explicitly. Also the transformations of the `parallel` directive by most of the OpenACC compilers are similar.

6 Related Work

The performance of NPB benchmarks are well studied for conventional multi-core processor based clusters. Results in [18] show that OpenMP achieves good performance for a shared memory multi-processor. Other related works also include NPB implementations of High-Performance Fortran (HPF) [15], Unified Parallel C (UPC) [2] and OpenCL [23]. Pathscale ported an older version of NPB (NPB 2.3) using OpenACC [9], but only SP and IS could be compiled and executed successfully. Moreover their implementation of the benchmark, IS, does not use the challenging bucket sorting algorithm; this algorithm poses irregular memory access pattern challenges that is not straightforward to solve. However, we do use this sorting algorithm and overcome the challenges by using

OpenACC's atomic and scan operation extensions. With high performance computing systems rapidly growing, hybrid programming models become a natural programming paradigm for developers to exploit hardware characteristics. Wu et al. [25] discuss a hybrid OpenMP + MPI version of SP and BT benchmarks. Pennycook et al. [21] describe the MPI+CUDA implementation of LU benchmark. The hybrid implementations commonly yield better performance if communication overhead is significant for MPI implementation and if computation for a single node is well parallelized with OpenMP. NAS-BT multi-zone benchmark was evaluated in [11] using OpenACC and OpenSHMEM hybrid model.

Grewe et al. [16] presented a compiler based approach that automatically translate OpenMP program to optimized OpenCL code for GPUs and they evaluated all benchmarks in NPB suite. Lee et al. [20] parallelized EP and CG from NPB suite using OpenACC, HMPP, CUDA and other models and compared the performance differences. But our implementation is different from theirs for these two benchmarks. Inspired by the similar subroutines of the benchmarks in NPB, Ding et al. [13] [12] developed a tool that can conduct the source code syntactic similarity analysis for scientific benchmarks and applications.

7 Conclusion

This paper discusses practices and optimization techniques for parallelizing and optimizing NAS parallel benchmarks for GPGPU architecture using the OpenACC high-level programming model. We present performance and speedup obtained by using an open source OpenACC compiler. We believe these techniques can be generally applicable for other programming models and scientific applications. We also analyze the effectiveness of these optimizations and measure their impact on application performance. Poorly selected options or using system default options for optimizations may lead to significant performance degradation. We also compared the performance of OpenACC NPB with that of the well-tuned OpenCL and CUDA versions of the benchmarks. The results indicate we achieve performance close to that of the well-tuned programs. This shows that using high-level programming directives and with the right optimization techniques, we are not only achieving the much needed portability but also achieving performance close to that of well-tuned programs. We also investigated and implemented the scan and cache optimizations in the compiler used. As future work, we will identify strategies to automate optimizations that we have used in our compiler for better programmability and perhaps performance.

References

1. NPB-CUDA. `http://www.tu-chemnitz.de/informatik/PI/forschung/download/npb-gpu/`, 2013.
2. NPB-UPC. `http://threads.hpcl.gwu.edu/sites/npb-upc`, 2013.
3. OpenACC. `http://www.openacc-standard.org`, 2013.
4. OpenCL Standard. `http://www.khronos.org/opencl`, 2013.

5. OpenMP. www.openmp.org, 2013.
6. 11 Tricks for Maximizing Performance with OpenACC Directives in Fortran. http://www.pgroup.com/resources/openacc_tips_fortran.htm, 2014.
7. CUDA. http://www.nvidia.com/object/cuda_home_new.html, 2014.
8. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2014.
9. Pathscale NPB2.3 OpenACC. https://github.com/pathscale/NPB2.3-OpenACC-C, 2014.
10. D. Bailey et al. The NAS Parallel Benchmarks. *NASA Ames Research Center*, 1994.
11. M. Baker, S. Pophale, J.-C. Vasnier, H. Jin, and O. Hernandez. Hybrid Programming using OpenSHMEM and OpenACC. In *First OpenSHMEM Workshop: Experiences, Implementations and Tools*, pages 74–89. Springer, 2014.
12. W. Ding, O. Hernandez, and B. Chapman. A Similarity-Based Analysis Tool for Porting OpenMP Applications. In *Facing the Multicore-Challenge III*, pages 13–24. Springer Berlin Heidelberg, 2013.
13. W. Ding, C.-H. Hsu, O. Hernandez, B. M. Chapman, and R. L. Graham. KLONOS: Similarity-Based Planning Tool Support for Porting Scientific Applications. *Concurrency and Computation: Practice and Experience*, 25(8):1072–1088, 2013.
14. R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A Hybrid Multi-core Parallel Programming Environment. In *Workshop on GPGPU*, 2007.
15. M. Frumkin, H. Jin, and J. Yan. Implementation of NAS Parallel Benchmarks in High Performance Fortran. *NAS Techinical Report NAS-98-009*, 1998.
16. D. Grewe, Z. Wang, and M. F. O'Boyle. Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems. In *CGO, 2013 IEEE/ACM International Symposium on*, pages 1–10. IEEE, 2013.
17. M. Harris, S. Sengupta, and J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. *GPU gems*, 3(39):851–876, 2007.
18. H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical report, NAS-99-011, NASA Ames Research Center, 1999.
19. S. Lee, D. Li, and J. S. Vetter. Interactive Program Debugging and Optimization for Directive-Based, Efficient GPU Computing, 2014.
20. S. Lee and J. S. Vetter. Early Evaluation of Directive-based GPU Programming Models for Productive Exascale Computing. In *SC 12*, pages 23:1–23:11. IEEE Computer Society Press, 2012.
21. S. J. Pennycook, S. D. Hammond, S. A. Jarvis, and G. R. Mudalige. Performance Analysis of a Hybrid MPI/CUDA Implementation of the NAS LU Benchmark. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):23–29, 2011.
22. 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.
23. S. Seo, G. Jo, and J. Lee. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. In *IEEE Intl Symp. on IISWC*, pages 137–148. IEEE, 2011.
24. X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, and B. Chapman. Compiling A High-Level Directive-based Programming Model for GPGPUs. In *Intl. workshop on LCPC 2013*, pages 105–120. Springer, 2014.
25. X. Wu and V. Taylor. Performance Characteristics of Hybrid MPI/OpenMP Implementations of NAS Parallel Benchmarks SP and BT on Large-Scale Multicore Clusters. *The Computer Journal*, 55(2):154–167, 2012.