Directive-Based Compilers for GPUs

Swapnil Ghike[1], Ruben Gran[2], Maria J. Garzaran[1], and David Padua[1]

[1] University of Illinois at Urbana-Champaign.Department of Computer Science
sghike2@illinois.edu garzaran@illinois.edu padua@illinois.edu
[2] University of Zaragoza. Departamento de Informatica e Ingenieria de Sistemas
rgran@unizar.es

Abstract. General Purpose Graphics Computing Units can be effectively used for enhancing the performance of many contemporary scientific applications. However, programming GPUs using machine-specific notations like CUDA or OpenCL can be complex and time consuming. In addition, the resulting programs are typically fine-tuned for a particular target device. A promising alternative is to program in a conventional and machine-independent notation extended with directives and use compilers to generate GPU code automatically. These compilers enable portability and increase programmer productivity and, if effective, would not impose much penalty on performance.
This paper evaluates two such compilers, PGI and Cray. We first identify a collection of standard transformations that these compilers can apply. Then, we propose a sequence of manual transformations that programmers can apply to enable the generation of efficient GPU kernels. Lastly, using the Rodinia Benchmark suite, we compare the performance of the code generated by the PGI and Cray compilers with that of code written in CUDA. Our evaluation shows that the code produced by the PGI and Cray compilers can perform well. For 6 of the 15 benchmarks that we evaluated, the compiler generated code achieved over 85% of the performance of a hand-tuned CUDA version.

Keywords: Directive-based compiler, OpenACC, GPGPU, evaluation, Cray, PGI, accelerator

1 Introduction

GPUs have started to play an important role in performance critical applications in computer graphics, scientific computing, gaming consoles and mobile devices, primarily because GPUs offer massive parallelism and computational power that is not available with the heavyweight multicore CPUs. GPUs have also shown better performance per watt in studies [15] conducted in the past using applications that represented varying domains and computational patterns. Many supercomputers including NCSA's Blue Waters supercomputer [31] at the University of Illinois employ GPUs. With the introduction of NVIDIA Tegra processor in tablet devices [27, 26], the applicability of heterogeneous architectures, i.e., architectures that combine a traditional general purpose CPU and specialized co-processors like the GPU, seems evident in all forms of computing.

As parallelism increases, memory bandwidth becomes a bottleneck. GPUs try to hide memory latencies by maintaining a large pool of thread contexts and switching between these contexts at virtually no cost. Thus, GPUs perform best when its multiprocessors are kept busy with hundreds or even thousands of threads that execute in parallel. However, extracting the best performance out of GPUs using APIs that are closer to hardware and C-like programming languages such as CUDA [28] and OpenCL [19] is a time consuming process for programmers. Correctness issues like avoiding deadlocks and race conditions, and performance issues like making optimal use of device memory bandwidth, avoiding cyclic CPU-GPU data movement and pipelining of asynchronous data movement have to be dealt with explicit instructions or algorithms by the programmer. Thus, increased performance comes at the cost of programmer productivity. On top of the existing programmer productivity drains, manually writing code using CUDA and OpenCL often leads to programs that are fine-tuned to perform only on a particular device. Performance portability thus becomes another major problem.

Loops are the natural candidates for translating to GPU kernels. A programmer can use CUDA or OpenCL to manually translate loops into GPU kernels. However, today a new programming paradigm for heterogeneous systems is emerging. Programmers can now use compilers to translate conventional CPU programs to heterogeneous code that is fine-tuned to the resources of the GPU being used. These compilers may need the programmer to provide hints or instructions in the form of directives. Providing directives in an OpenMP-like manner usually takes less efforts than writing a CUDA or OpenCL program. The heterogeneous code can then offload loops to the GPU and take care of host-GPU data communication. This approach has the potential to obtain the best of both worlds - good performance, if not the best as that obtained with optimized CUDA or OpenCL programs, and increased programmer productivity and code portability.

The goal of this work is to evaluate the limitations of the PGI [30] and Cray [9] compilers to generate efficient accelerator codes when using compiler directives: PGI programming model's proprietary directives and OpenACC [5], respectively. For the version of PGI compiler that we used, the OpenACC directives were not available. More recent versions of the PGI compiler support the OpenACC directives, and it has been shown that heterogeneous programs using the PGI directives have the same performance as those using the PGI compiler with the OpenACC directives [22]. The paper makes the following contributions:

1. We first evaluate standard loop transformations developed for autoparallelization, such as distribution and privatization. Our experiments show that the compilers studied applied few of the studied transformations - loop alignment, distribution, interchange, privatization, reversal and skewing. Furthermore, the compilers were able to generate optimized code for reduction, but not for prefix scan or histogram reduction loops.
2. We present a comprehensive set of the transformations that programmers should apply so that compilers can generate efficient code for accelera-

tors. For this study we applied a sequence of manual transformations to 15 OpenMP programs from the Rodinia benchmark suite to convert them to a form that can be understood by the PGI and Cray compilers. Our results show that with the Cray compiler, fewer manual transformations were needed in the case of 8 out of 15 benchmarks. We also demonstrate the impact of each transformation on the performance of the heterogeneous programs.

3. We assess the performance obtained by these compilers. We compare the performance of the heterogeneous versions of Rodinia benchmarks compiled by the PGI and Cray compilers with that of the manually-optimized OpenMP and CUDA versions of those programs. Our results show that the code generated by the compilers achieves more than the 85% of the performance of 6 out 15 manually optimized versions of Rodinia, and perform worse for other benchmarks depending on various factors such as data communication between the host and the device, non-coalesced device memory accesses, etc.

The rest of this paper is organized as follows: Section 2 briefly discusses related work; Section 3 describes experimental framework; Section 4 is an evaluation of the abilities of these compilers to analyze dependences and perform loop transformations; Section 5 presents the transformations required to convert OpenMP programs to their heterogeneous versions and the impact of each transformation on their performance; Section 6 evaluates the performance of Rodinia benchmarks when compiled with the PGI and Cray compilers and discusses the performance bottlenecks; and finally Section 7 concludes.

2 Related Work

GPUs are an attractive platform for scientific simulation and general purpose applications that can exploit the data parallelism they provide. However, programming and tuning for these devices is onerous [2, 11]. Compilers such as CUDA [28] and OpenCL [19] aim at easing the low level programming of GPUs, but they expose the GPU architecture (including the different memory domains and the SPMD/SIMD architecture) to the programmer, which makes a program implementation hardware dependent. This also increases the learning and development phase, what makes the programming task more error-prone.

Cray [9], CAPS [4], and PGI [30] have recently introduced a high-level programming paradigm, similar to that of OpenMP [29], called OpenACC [5]. In this paradigm, the programmer uses pragmas or directives to annotate the sequential code and specify the loops and regions of code that can be offloaded from a host CPU to a GPU, as well as the data to be transferred.OpenMP4.0 [29] has also recently introduced extensions to support accelerators.

hiCUDA [12] sits between the two extremes described above. It relies on pragmas, but it does not ease the programmer task as much as the Cray, CAPS or PGI compilers. Other experimental compilation tools like CGCM [16] and PAR4ALL [1] aim at automating the process of CPU-GPU communication and the detection of the pieces of code that can run in parallel. The work by Lee and

Eigenmann [20] proposes OpenMPC, an API to facilitate translation of OpenMP programs to CUDA, and a compilation system to support it.

Previous studies, like [8, 24, 13, 17, 32, 14, 22, 10] also evaluate directive-based compilers that generate code for accelerators. The main difference is that this work covers more programs and includes a study of transformations. While some of the previous works have performed assessments similar to those presented in this paper, they typically focus on one or two applications or on benchmarks sets containing small and regular kernels, such as Polybench. In this paper we evaluate the Rodinia v2.0.1. benchmark suite [6, 7], which contains regular and irregular applications from different domains. Also, this paper evaluates two of the three currently available commercial compilers (CAPS is the only other commercial compiler that we did not consider in this work).

While we perform our evaluation using the Rodinia benchmark suite, which contains applications from different domains, most previous works only experiment with 1 or 2 applications, except for the project discussed by Grauer et al. [10] and Lee and Vetter [22]. The work by Grauer et al. uses the PolyBench collection which contains regular kernels mostly from the linear algebra domain. We intend not to limit our results to a reduced type of applications and we show results for a larger variety of benchmarks, in an effort of covering the real world application spectrum, including irregular benchmarks like PathFinder for Dynamic Programming and Breadth First Search for Graph Traversal. The work by Lee and Vetter [22] is the most comprehensive with respect to the number of applications and compilers evaluated. Lee and Vetter evaluate 8 Rodinia benchmarks (out of the 15 we evaluate) and some scientific kernels, such as Jacobi or kernels from the NAS benchmarks. They also evaluate the PGI, CAPS, OpenMPC [20, 21], and R-Stream [23] compilers. However, the main difference with this study, is that the work reported here also includes the transformations steps that programmers must follow to transform OpenMP programs into directive-based programs so that these compilers can generate efficient accelerator code. In our work we also evaluate whether the studied compilers perform standard transformations, such as loop distribution or privatization to enable parallelism. We have not found a similar study on previous works. Only the work by Grauer et al. [10] evaluates the use of three of the CAPS-specific pragmas (permute, unroll, and tiling) intended to drive code transformations for code optimization. However, this work has a much more limited scope than ours.

3 Environmental Setup. Target Platform and Compilers

Our experiments were performed using a single Cray XK6 node. Table 1 presents characteristics of this experimental hardware platform, and Table 2 shows for each compiler, the compiler version and flags used in our experiments.

We evaluated two directive-based compilers for heterogeneous systems: PGI [30] and Cray [9] compilers. With the Cray compiler we used the OpenACC [5] directives, while with the PGI compiler we used the PGI programming model's proprietary directives [30]. For the version of PGI compiler that we used, the

Table 1. Hardware characteristics

Type	CPU	GPU
Name	AMD Opteron 6200	NVIDIA Tesla X2090
Cache	L2/L3 - 512kB/12MB	L2(shared) 42Kb(Multiprocessor)
# cores	16 cores	512 CUDA cores
Peak Perf.	294.4GFLOPS	1331 GFLOPS
Frequency	2.1 GHz	1.3GHz

Table 2. Compilation flags for each compiler

Specification/flags	PGI	CRAY XK	NVCC/GCC
Version	11.10-0	CCE v 8.1.0.139 PrgEnv-cray v 4.0.46	4.0.17a / 4.3.4
Baseline optimizations	-fast -fastsse -O3	-O3	-O3 / -O3 -Ofast
Platform optimizations	-tp=bulldozer-64	-h cpu=interlagos	-arch=sm_20 -march=bdver1
Sequential flags	None	-h noacc -h noomp -h noomp_acc	NA NA
OpenMP flags	-mp=allcores	-h noacc -h noomp_acc	NA
Heterogenous flags	-ta=nvidia,keepgpu	-h noomp_acc	NA
Compilation Report	-Minfo	-hlist=m	NA

OpenACC directives were not available. More recent versions of the PGI compiler support the OpenACC directives, but it has been shown that heterogeneous programs using the PGI directives have the same performance as those using the PGI compiler with the OpenACC directives [22]. The PGI and the OpenACC directives are semantically equivalent (as they both enable the same functionality), but they differ syntactically.

3.1 Benchmarks

We used two benchmark suites. First, a micro-benchmark suite that we developed for this project and the other is the Rodinia benchmark suite 2.0.1 [6, 7]. Our micro-benchmark suite consists of a set of loops to assess the ability of the compilers to perform standard loop transformations for exposing parallelism. These micro-benchmarks were designed to verify if the PGI and Cray compilers can perform loop transformations that remove loop carried dependences to expose parallelism.

In the experimental section, we used the Rodinia benchmark suite 2.0.1 [6, 7]. Out of the 16 benchmarks available in the Rodinia benchmark suite, we have used 15. We did not include Mummer-GPU in our analysis because of the limitations of using unions in the compilers that we studied. These applications are provided with two different implementations, OpenMP and CUDA. The CUDA implementation exploit the different types of memories available in the GPU: global, shared, constant and texture memories.

3.2 Performance Measurement

Our time measurement excludes any setup and initial I/O performed by the program. In case of CUDA programs and heterogeneous programs, we also measure the time spent in three separate components of the program using the CUDA profiler [25]: time spent in the GPU kernels, time taken to transfer data between CPU and GPU, and the summation of the time spent in any sequential computation that is an integral part of the program and the overhead of launching GPU kernels and initiating data transfers.

Table 3. PGI and Cray: loop transformations to micro-kernels

Transformation	Improves GPU kernel	Speedup		Conclusions	
		PGI	Cray	PGI	Cray
Alignment	Yes	299.14	33.73	No	No
Distribution	Yes	212.22	15.59	No	No
Interchange	No, threads spawn once	1.00	1.00	Not needed	
Privatization	Yes	0.99	1.00	Yes	Yes
Reversal	Yes	13.59	30.42	No	No
Skewing	No, skewed iteration space	0.72	0.28	No	No

4 Micro-Kernels

In this section, we report the results of our analysis that determines the effectiveness of the PGI and Cray compilers in automatically applying standard transformations that remove cross-iteration dependences that prevent a loop from being parallelized. For this analysis, we wrote a set of 9 loop nests. In 6 of the loop nests, application of a single transformation among alignment, distribution, interchange, privatization, reversal and skewing will allow iterations of that loop nest to run in parallel with each other using multiple threads [18]. The other 3 loop nests perform reduction, prefix scan, and histogram reduction operations, and can be parallelized on a GPU by changing the algorithm.

Table 3 shows the first 6 loops. The first column shows the name of the transformation; the second column states whether the transformation enables a loop nest to efficiently use the GPU.

We used the compiler reports to determine if the compiler applied the transformations automatically. However, since these reports were not always clear, for each case we executed both the original and the transformed loop nests and recorded their execution times. The original loop nest needs to be transformed to be parallelizable. If the compiler is not able to automatically apply the transformation, then the transformed version of the loops executes significantly faster than the original version. The third column of Table 3 shows the speedups of the transformed versions of the loop nests over the original loop nests. The last column of Table 3 shows our conclusions on whether the compilers were able to automatically perform the corresponding transformation.

Table 3 shows that out of all the transformations, the ones that enable a loop to make efficient use of GPU parallelism are alignment, distribution, privatization and reversal. Interchange and skewing are not helpful in improving performance on a GPU.

Loop interchange can improve performance when an outer loop executes sequentially with an inner parallelizable loop. If the inner loop is parallelized, each outer iteration spawns and joins the multiple threads executing the inner loop. In this case, the inner parallel loop can be interchanged with the sequential outer loop as long as no dependences are violated, thereby spawning threads only once. However, this transformation is not necessary to parallelize loops on a GPU while using the PGI and Cray compilers, because the PGI and Cray compilers split the inner parallel loop across multiple threads and execute all the iterations of the sequential outer loop in each thread (redundantly). The skewing transformation is not suitable for GPUs, because the code that results

Table 4. PGI and Cray: automatically parallelizing computational patterns

Idiom	Conclusions	
	PGI	Cray
Reduction	Yes	Yes
Prefix Scan	No	No
Histogram	No	No

after applying the transformation executes in a skewed iteration space, that leads to warp divergence.

We also wrote 3 loop nests that perform reduction, prefix scan and histogram reduction. These computational patterns, in particular reduction, are used often among scientific applications and they can be parallelized only by a change of algorithm. However, compilers can potentially recognize these patterns and produce the parallelized algorithms automatically. When the compilers do not recognize the computation patterns, the generated code results in a naive GPU implementation that executes serially and obtains low performance. As there is no equivalent parallelizable sequential loop nests for these patterns, these computational patterns can only be parallelized in a programming language like CUDA or OpenCL that gives the programmer a fine grained control of the code executed by each thread and memory barriers. The first column of Table 4 shows the transformation; the second column shows whether the compilers were able to automatically parallelize the computational patterns. Table 4 shows that none of the compilers was able to parallelize prefix scan and histogram reduction automatically. In such cases, providing a finer control over parallelism would benefit programmers when the automatic recognition/parallelization of computational patterns does not succeed.

In the second part of this work, we study 15 benchmarks from the Rodinia suite. We observed that in these benchmarks loop distribution, privatization, reduction, and prefix-scan optimization transformations were necessary. The other loop transformations were not applicable , but might be useful for a different set of applications.

5 Transformation Steps

We have used the Rodinia benchmarks to assess the transformations that were necessary to convert the OpenMP programs to a format that the PGI and Cray directive-based compilers can understand and compile to produce device kernel code. Section 5.1 describes the transformations steps that we had to follow to transform programs. Section 5.2 describes the reasons for applying each transformation and the performance impact of each transformation.

5.1 Transformations

While most previous works discuss the individual transformations applied to each benchmark, in this Section we have tried to synthesize the overall strategy

that we followed to transform all the OpenMP Rodinia benchmarks. Next, we describe the transformations in the order in which were applied:
T1: Convert the program to C99 from C++, if using the PGI compiler. The version of PGI compiler used did not allow C++ code in the parallel regions.
T2: Insert *parallel regions*, in order to delimit parallel loops.
T3: *restrict* attribute, PGI explicitly requires removing array aliasing.
T4: Convert a multi-dimensional array to an array stored in contiguous memory. When GPU kernels work with sub-dimencions of a multi-dim array, host-device communication must be conscious of multi-dim array memory mapping. Flattening these arrays improves these communication tasks.
T5: Remove pointers to arrays in structures and stop the use of unions. T5 is applied because the compiler cannot de-reference the pointers to the arrays in the structure to perform the data transfer between CPU and GPU. Similarly, the compilers could not correctly allocate space on the GPU when unions were used inside the parallel regions [3].
T6: Inline procedures. The PGI compiler required all the procedures inside a parallel region to be manually inlined. The Cray compiler could inline procedures with primitive data type arguments.
T7: Add *data clauses* to *parallel regions*. Both compilers were able to automatically generate a CPU-GPU memory copy command to transfer data between CPU-GPU when the array size is known at compilation time. Otherwise, the programmer has to manually specify the array size in data clauses.
T8: Use the independent clause. The programmer can enable compiler parallelization using the independent clause to inform the compiler that the following loop does not carry dependences across iterations. Reasons:

a. The loop iterations have an output dependence, but parallelizing the loop results in a benign data race in which threads write the same value to a given memory location (BFS).
b. The compiler detects false dependences between iterations due to array index calculations that involve runtime variables, whose values are unknown to the compiler. Programmers can assert that there are no data dependences (NW).
c. Compilers could not analyze the array index calculations.(PGI: KM, BP, HS, LUD, CFD, LC, PF, HW; Both: SRAD)
d. The Cray compiler generated incorrect GPU code due to a bug. As a workaround we used the independent clause. This bug will likely go away in the coming compiler releases.(KM, BP, LUD, SC, LC, PF, HW)

T9: Insert *data regions*. They help to avoid cyclic data movement between the CPU and GPU. The programmer can specify the variables or arrays to be copied into the GPU memory at the entry point of the data region and the variables or arrays to be copied out at the exit point of the data region.
T10: Change the size and/or number of parallel regions with respect to the OpenMP code. Exploiting parallelism in tightly nested loops is crucial

[3] In the PGI version of CFD Solver, we also had to separate the individual float values included in a structure, but this was most probably due to a bug.

Table 5. Summary of directives and transformations applied in 15 Rodinia Benchmarks

Directives and Transformations			Total		Others
			PGI	Cray	
Directives. Programmer applied		Parallel Regions (T2)	15	15	0
		Data Clausues (T7)	13	14	0
		Independent (T8)	11	10	0
		T8.a	1	1	0
		T8.b	1	1	0
		T8.c	9	1	0
		T8.d	0	7	0
		Data Regions (T9)	14	14	0
		Collapse (T11)	0	6	0
Transformations	Compiler Applied	Privatization	15	15	0
		Reduction	5	5	6
	Programmer Applied	C++ to C99 (T1)	5	0	0
		restrict (T3)	15	0	0
		multi to single (T4)	3	3	0
		remove pointers (T5)	6	5	0
		inline (T6)	9	1	0
		distribution (T10)	11	11	0
		T10.a	2	2	0
		T10.b	3	3	0
		T10.c	3	3	0
		T10.d	3	3	0
		alignment	0	0	0
		reversal	0	0	0
		skewing	0	0	0
		prefix-scan	0	0	1
		histogram	0	0	0

to obtaining high performance. Also, expanding the parallel regions, changing the loops that are parallelized etc. can help in obtaining higher performance. Our results show that this is an important step that requires manual effort and manual tuning from the programmer. Reasons:

a. Distribute a parallel loop over an inner loop if the inner loop is parallel and contains sufficient amount of computation. The resulting loop is a good candidate for GPU computation. For some applications distribution has to be applied after procedure inlining.
b. Remove the reduction from an accelerator region to facilitate compiler optimization of the remaining code. The reduction can be performed in a separate parallel region or on the CPU depending on the amount of computation and the amount of CPU-GPU data communication involved.
c. Change the boundaries and/or number of the parallel regions. Depending on the availability of statements that can execute in parallel, we can expand the parallel regions to include more computation in them and/or generate new parallel regions. On the contrary, we should not run in GPU a loop that has a small amount of computation and requires a large amount of CPU-GPU transfer.
d. Change loops that are parallelized while maintaining the same number of parallel regions. This is achieved by interchanging loops or by merely changing the position of parallel directive if the loops are loosely nested and interchanging those loops is not possible without substantial manual effort.

T11: Use collapse clause, for CRAY only (PGI does automatically). Using a collapse clause leads to an increase in multiprocessor occupancy which is a ratio of the number of threads executing on the GPU at a given time to the maximum number of threads that could execute on that GPU. This may increase the utilization of the GPU parallelism and reduce the execution time.

Table 5 classifies these transformations as insertion of compiler directives and other transformations. Compiler directives must always be inserted by the programmer, the other program transformation can be classified based on whether

they are applied by the compiler or the programmer. The Column Total shows for how many benchmarks each directive or transformation was applied when using the PGI and the Cray compilers. Column Others shows the number of times a transformation could have been applied, but was not applied for reasons that will be explained next.

With respect to transformations, privatization and reduction are the only two transformations that were automatically applied by the compilers. Privatization was needed in all the benchmarks and both compilers applied it successfully. Both compilers applied reduction to 5 benchmarks (BP, BFS, LMD, PF, SC), but both failed to take advantage of it for 6 benchmarks. When the reduction appears alone in a single loop both compilers can recognize it. However, when several reductions appear in a single loop or the reduction is intermixed with other computations in the same loop, the compilers usually fail to recognize it. This occurred in 6 benchmarks (CFD, HW, KM, LUD, SRAD, SC), as shown in the column Others. When we manually distributed the loops to isolate the reduction in a single loop the compiler recognized it.

For the PGI compiler, the use of the restrict keyword and procedure inline was needed for 15 and 9 benchmarks, respectively. With the Cray compiler we did not have to use the restrict keyword and inlining is necessary only for one benchmark. PGI also required to transform the codes from C++ to C99. With respect to the other directives and transformations, both compilers perform very similar. Therefore, we conclude that Cray requires less effort than PGI.

The transformations alignment, reversal, skewing, and histogram were not needed for these benchmarks although they could be helpful for other benchmarks. Prefix scan was present in HW, it was not recognized (Section 4).

5.2 Impact of Each Transformation on Performance

Table 6 shows the performance impact of the previous transformation steps on the Rodinia benchmarks for the PGI and Cray compilers. Each row in the table specifies the last transformation that has been applied and the the slow-ness in performance with respect to the performance obtained after applying all transformations in Section 5.1 (step T10 for PGI and T11 for Cray). In the same column, the same slow-ness means that the transformation was either not needed or it had no effect. We show results only after applying T7, because heterogeneous compute regions can execute in the GPU in most cases only after T7 has been applied. To assess the impact on performance of a transformation we need to compare two consecutive rows. Thus, to asses the impact of T8 we need to compare the data in row T7 with the data in row T8. Notice that the Cray compiler was still under development and some benchmarks (PF, LC, HW, LMD and SC) could not be compiled correctly (shown as CF in Table 6) due to internal compiler bugs. In some cases, executing a program version took so long, that we decided to stop the execution and label it as ∞.

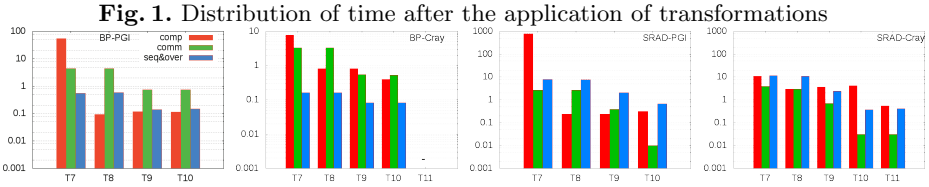
In Figure 1, we show results for BP and SRAD (the rest are not shown due to space limitations) with both compilers. Each chart shows for each transformation step, the breakdown of the program execution time into its three components

Table 6. Relative slow-ness after each step with the PGI and Cray compilers.

PGI	PFDR	KM	BFS	BP	HS	LUD	NN	NW	SRAD	SC	CFD	LC	PF	HW	LMD
T10	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
T9	1.0	100.5	2.5	1.0	1.0	1.0	1.0	1.0	2.7	∞	1.0	∞	1.0	∞	∞
T8	5.3	102.4	6.7	5.0	13.7	537.1	1.0	∞	10.6	∞	14.2	∞	3.3	∞	∞
T7	5.3	∞	128.3	59.4	∞	5859.7	1.0	∞	858.6	∞	14.2	∞	3.6	∞	∞

Cray	PFDR	KM	BFS	BP	HS	LUD	NN	NW	SRAD	SC	CFD	LC	PF	HW	LMD
T11	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	CF	CF	CF	CF
T10	1.0	1.0	1.0	1.0	4.5	1.0	1.0	1.0	4.7	1.0	1.0	CF	CF	CF	CF
T9	1.0	∞	2.7	1.5	4.5	1.0	1.0	1.0	6.7	CF	1.0	CF	CF	CF	CF
T8	350.8	∞	7.5	4.3	34.1	20.4	1.0	∞	19.2	CF	50.6	CF	CF	CF	CF
T7	350.4	∞	170.2	11.2	34.1	21.1	1.0	∞	26.3	CF	50.6	CF	CF	CF	CF

Fig. 1. Distribution of time after the application of transformations

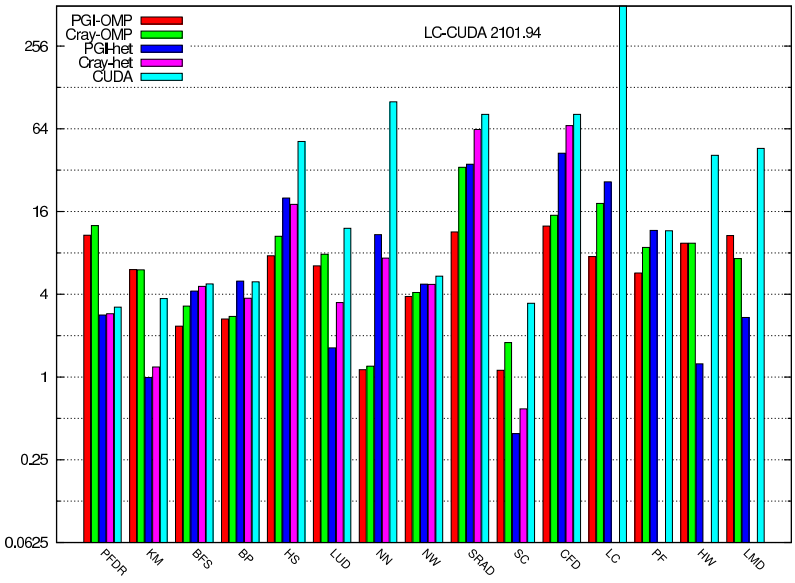


GPU-comp	Time taken by device kernel execution
CPU-GPU-comm	Time required for CPU-GPU data transfer
Seq&Over	Time taken by sequential computation on the CPU plus any overhead of launching GPU commands over the PCIE bus
-	The transformation was not applied

- device kernel execution time in the GPU, time of CPU-GPU data communication, and the time of sequential computation and any overhead of launching the device commands over the PCIE bus. The bars for the three components of each program are plotted such that the sum of these three components (which equals the overall program time) represents the slow-ness with respect to the performance obtained after applying all transformations in Section 5.1. Thus, the three components of transformations T10, in case of PGI, and T11, in case of Cray, sum up 1 (note the logarithmic scale). In the charts, label - states that the transformation was not applied/needed. As expected, performance improves as more transformations are applied to each benchmark. From Figure 1 and Table 6, we can observe that there is not a single transformation responsible of the whole performance improvement. For each benchmark, each transformation step has a varying degree of impact on the performance.

The impact of each transformation on the individual components (kernel, communication and seq. and overhead time) of the overall program time can be seen in Figure 1. The application of T8 reduces the time spent in device kernel execution. The application of T9 reduces the time spent in CPU-GPU data communication. The application of T10 has an effect on all three components. The magnitude of these components can increase or decrease depending on which combination among T10.a, T10.b, T10.c and T10.d is applied and how effective each of these transformation is for that program. Finally, the application of T11 reduces the time spent in device kernel execution.

Fig. 2. Speedups over sequential programs compiled with PGI



6 Performance of Rodinia Benchmarks

In this section, we evaluate the performance of five versions of the Rodinia benchmark suite: OpenMP, sequential, CUDA and two heterogeneous versions (following the steps in Section 5). For each benchmark, the suite provides two versions - an OpenMP version and an optimized CUDA version [6, 7]. One heterogeneous version uses OpenACC directives (Cray compiler), while the other uses the PGI programming model directives (PGI compiler).

Figure 2 shows speedup of the OpenMP, heterogeneous, and CUDA versions against the performance of the sequential version compiled with the PGI compiler. Since PF, LC, HW, and LMD could not be correctly compiled by the Cray compiler, the corresponding spaces in the figure have been left blank. The comparison of heterogeneous versions and the CUDA counterparts will help us to identify the performance bottlenecks and understand the effectiveness of the PGI and Cray compilers in exploiting the GPU resources.

For a total of 6 benchmarks (PFDR, BFS, BP, NW, CFD, PF) at least one of the heterogeneous versions managed to reach a performance over the 85% of the CUDA performance. In fact, three benchmarks were able to achieve close to 100% of the CUDA performance (BFS, BP, and PF). Even more, in some cases the heterogeneous version performs slightly better than the CUDA version. This is due to T10 (Section 5) transformation which parallelizes portions of code that were not formerly running in parallel. However, 8 benchmarks (KM, HS, LUD, NN, SC, LC, HW, LMD) obtain less than 50% of the CUDA performance.

The poor performance of the heterogeneous programs is due to issues such as poor computation to data communication ratio, excessive device global memory accesses, non-coalesced memory accesses and inefficient use of shared memory etc. We briefly explain the reasons behind the poor performance achieved by individual heterogeneous programs as compared to their CUDA versions.

The heterogeneous versions of KM suffer from both high amount of data transfer and non-coalesced memory accesses in the GPU kernels, hence they produce slow performance. In the case of HS, the heterogeneous versions perform a data copy within the GPU as part of the stencil computation, whereas CUDA achieves the same effect by a pointer swap. Thus, the heterogeneous versions of HS shows reduced performance. LUD involves triangular matrix operations and has many row-wise and column-wise dependencies [7]. The heterogeneous versions make a lot of non-coalesced accesses to the device memory since the device cache or shared memory is not big enough to hold the entire matrix involved in the computation, which leads to reduced performance in GPU computation. The CUDA version of LUD implemented tiling which improves the cache performance. The heterogeneous versions of NN do not parallelize enough part of the kernel, thereby leading to a slower performance compared to the CUDA version. The heterogeneous versions of SC suffer from cyclic CPU-GPU data communication and thus become inefficient. The extraordinary performance of LC with CUDA is obtained by using a technique called persistent blocking [6, 3] wherein all the iterations are performed in a single device kernel call and all cells are processed concurrently with one thread block allocated to each cell. In the kernel of heterogeneous HW, each thread loads data from memory in a loop which causes a lot of non-coalesced global memory loads. The number of misses in L2 cache of the GPU in the heterogeneous version are also about 48X as compared to the CUDA version. In case of LMD, heterogeneous version perform very little computation for the number of device global memory accesses it performs, which results in bad performance.

Focusing on OpenMP vs. heterogeneous versions will give a comparative perspective of the speedups that could be obtained using directive-based parallel programming techniques that exploit parallelism from the CPU and the GPU. We can observe that for 9 benchmarks, at least one heterogeneous version has performed better than its corresponding OpenMP version. However there are certain benchmarks whose heterogeneous versions do not perform as well. The slow performance of the heterogeneous versions of KM, SC, LUD, HW, and LMD has been already explained. In case of PFDR, the CPU-GPU data communication itself in both the CUDA and heterogeneous versions takes more time than the whole OpenMP program, so communication/computation ratio is quite bad for GPU approach.

7 Conclusions

In this paper, we have evaluated the effectiveness of two directive-based compilers, PGI and Cray, in: i) dependence analysis and automatic application of

transformations ii) the programming effort to transform OpenMP programs to heterogeneous versions iii) performance as compared to OpenMP and CUDA.

From the dependence analysis, we found that out of the 6 transformations and 3 idioms studied, both compilers were only able to privatize and generate code for reduction. Thus, the PGI and Cray compilers still seem to rely on the programmer to expose the parallelism.

We have proposed a sequence of steps to transform OpenMP programs to their heterogeneous versions. In this analysis, we observed that the Cray compiler was able to automatically inline most functions inside the parallel regions as opposed to the PGI compiler, which led to considerably reduced programmer efforts. The difficulty of programming with the directive-based compilers is closer to OpenMP programming than to CUDA programming.

In terms of performance, the versions compiled Cray performed faster than the ones of PGI compiler for 8 out of 15 Rodinia benchmarks. In comparison to fine-tuned CUDA versions, 6 out of 15 heterogeneous versions ran over the 85% of the CUDA performance. This shows the potential of these heterogeneous directives-based compilers to produce efficient code and at the same time increase programmer productivity. In comparison to OpenMP, heterogeneous versions of the code ran faster in 9 out of 13 benchmarks at a similar programming effort to OpenMP. Degradation in performance of heterogeneous versions occurred because the compilers produced inefficient code and we could not overcome such inefficiencies due to not being able to express low-level optimizations using the directive-based compilers in the way CUDA versions did.

8 Acknowledgments

This research is part of the Blue Waters sustained-petascale computing project, which is supported by NSF (award number OCI 07-25070) and the state of Illinois. This research was supported in part by NSF under Award CNS 1111407. It was also supported by grants TIN2007-60625, TIN2010-21291-C02-01 and TIN2013-64957-C2-1-P (Spanish Government and European ERDF), gaZ: T48 research group (Aragon Government and European ESF).

References

1. Amini, M., et al.: Static compilation analysis for host-accelerator communication optimization. In: LCPC (2011)
2. Bordawekar, R., Bondhugula, U., Rao, R.: Can cpus match gpus on performance with productivity?: Experiences with optimizing a flop-intensive application on cpus and gpu. Tech. Rep. RC25033, IBM (August 2010)
3. Boyer, M., Tarjan, D., Acton, S.T., Skadron, K.: Accelerating leukocyte tracking using cuda: A case study in leveraging manycore coprocessors. In: Proc. of IPDPS. pp. 1–12 (2009)
4. CAPS Enterprise: Hmpp workbench (2011), `http://www.caps-entreprise.com/technology/hmpp/`
5. CAPS Enterprise and Cray Inc. and NVIDIA and the Portland Group: The openacc application programming interface, v1.0 (November 2011), `http://http://www.openacc-standard.org/`
6. Che, S., et al.: Rodinia: A benchmark suite for heterogeneous computing. In: IISWC 2009. pp. 44 –54 (oct 2009)
7. Che, S., et al.: A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In: Proc. of IISWC. pp. 1–11 (2010)

8. Cloutier, B., Muite, B.K., Rigge, P.: A comparison of cpu and gpu performance for fourier pseudospectral simulations of the navier-stokes, cubic nonlinear schrodinger and sine gordon equations. ArXiv e-prints (jun 2012)
9. CRAY: Cray Compiler Environment (2011), http://docs.cray.com/books/S-2179-52/html-S-2179-52/index.html
10. Grauer Gray, S., Xu, L., Searles, R., Ayalasomayajula, S., Cavazos, J.: Auto-tuning a high-level language targeted to gpu codes. In: Proc of InPar. pp. 1–10 (2012)
11. Hacker, H., Trinitis, C., Weidendorfer, J., Brehm, M.: Considering gpgpu for hpc centers: Is it worth the effort? In: Facing the Multicore-Challenge. pp. 118–130 (2010)
12. Han, T., Abdelrahman, T.: hicuda: High-level gpgpu programming. Parallel and Distributed Systems, IEEE Transactions on 22(1), 78 –90 (jan 2011)
13. Henderson, T., Middlecoff, J., Rosinski, J., Govett, M., Madden, P.: Experience applying fortran gpu compilers to numerical weather prediction. In: Proc. of SAAHPC. pp. 34 –41 (july 2011)
14. Hernandez, O., Ding, W., Chapman, B., Kartsaklis, C., Sankaran, R., Graham, R.: Experiences with high-level programming directives for porting applications to gpus. Lecture Notes in Computer Science 7174, 96–107 (2012)
15. J. Enos and et al.: Quantifying the impact of gpus on performance and energy efficiency in hpc clusters. In: Internatioanl Green Computing Conference. pp. 317 –324 (Aug 2010)
16. Jablin, T.B., et al.: Automatic cpu-gpu communication management and optimization. SIGPLAN Not. 47(6), 142–151 (Jun 2011)
17. Jin, H., Kellogg, M., Mehrotra, P.: Using compiler directives for accelerating cfd applications on gpus. Lecture Notes in Computer Science 7312, 154–168 (2012)
18. Kennedy, K., Allen, J.R.: Optimizing compilers for Modern Architectures: a Dependence-Based Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2002)
19. Khronos Group: Opencl - the open standard for parallel programming of heterogeneous systems (2011), http://www.khronos.org/opencl
20. Lee, S., Eigenmann, R.: Openmpc: Extended openmp programming and tuning for gpus. In: Proc. of SC10 (2010)
21. Lee, S., Min, S.J., Eigenmann, R.: Openmp to gpgpu: A compiler framework for automatic translation and optimization. In: Proc. of PPoPP'09 (2010)
22. Lee, S., Vetter, J.S.: Early evaluation of directive-based gpu programming models for productive exascale computing. In: Proc. of SC12 (2012)
23. Leung, A., Vasilache, N., Meister, B., Baskaran, M., Wohlford, D., Bastoul, C., Lethin, R.: A mapping path for multi-gpgpu accelerated computers from a portable high level programming abstraction. In: Proc. of GPGPU (2010)
24. Membarth, R., Hannig, F., Teich, J., Korner, M., Eckert, W.: Frameworks for gpu accelerators: A comprehensive evaluation using 2d/3d image registration. In: Proc of SASP. pp. 78 –81 (june 2011)
25. Nvidia: Compute Command Line Profiler. NVIDIA Whitepaper
26. Nvidia: The Benefits of Multiple CPU Cores in Mobile Devices. NVIDIA Whitepaper, {http://www.nvidia.com/content/PDF/tegra_white_papers/Benefits-of-Multi-core-CPUs-in-Mobile-Devices_Ver1.2.pdf}
27. Nvidia: Bring high-end graphics to handheld devices. Nvidia White Paper (2011), {http://www.nvidia.com/content/PDF/tegra_white_papers/Bringing_High-End_Graphics_to_Handheld_Devices.pdf}
28. Nvidia Corporation: Nvidia cuda programming guide version 4.0 (2011), http://developer.download.nvidia.com
29. OpenMP: Openmp: Complete specification v4.0 (2013), http://openmp.org/wp/resources/
30. The Portland Group: Pgi compiler reference manual (2011), http://www.pgroup.com/doc/pgiref.pdf
31. Website, B.W.: http://www.ncsa.illinois.edu/bluewaters/ (2011), {http://www.ncsa.illinois.edu/BlueWaters/}
32. Wienke, S., Springer, Paul adn Terboven, C., an Mey, D.: Openacc: First experiences with real-world applications. In: Proc. of EuroPar (2012)