

Compiler and Runtime Based Parallelization & Optimization

Guray Ozen and Eduard Ayguadé, Jesús Labarta
Universitat Politècnica de Catalunya - UPC

ABSTRACT

Graphics Processing Units (GPU) have been widely adopted to accelerate the execution of HPC workloads due to their vast computational throughput. Developing applications able to exploit the dazzling performance of GPUs is not a trivial task, and becomes even harder when they have irregular data access patterns or control flows. Several approaches have been proposed to help simplify accelerator programming. Models like OpenACC and OpenMP are intended to solve the aforementioned programming challenges. They take a directive based approach which allows the users to insert non-executable directives that guide the compiler to handle the low-level complexities of the system. However they have a performance gap with native programming models as their compiler does not have comprehensive knowledge about how to transform code and what to optimize. This thesis targets directive-based programming models to enhance their capability for GPU programming.

ACM Reference Format:

Guray Ozen and Eduard Ayguadé, Jesús Labarta. 2018. Compiler and Runtime Based Parallelization & Optimization. In *Proceedings of Supercomputing (SC) (SUPERCOMPUTING'18)*. ACM, New York, NY, USA, 2 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 THESIS CONTRIBUTIONS

In this thesis, we present several contributions in the field of directive based programming models, compilers and runtime algorithms aimed at leveraging the exploration of GPUs.

1.1 Task based GPU Offloading Model

We developed our MACC infrastructure which is a new task-based programming model as a dialect of a combination of OpenMP and OmpSs. It is developed on top of the Mercurium C/C++ source-to-source compiler. Figure 1 shows the compilation phases. Our aim is to let the compiler offload and parallelize code regions automatically under the control of user directives.

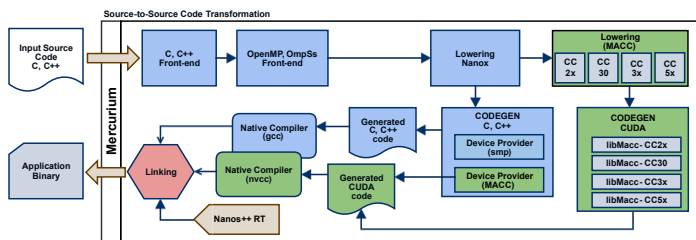


Figure 1: Compilation phases of MACC Compiler.

SUPERCOMPUTING'18, November 2018, Dallas, Texas USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . \$15.00
<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

```

1 #pragma omp target device(cuda)
2 #pragma omp task inout(C) out(D) TASK 1
3 __global__ void cuda-kernel(double C, double D) {
4     cuda-codes
5 }
6 int main(int argc char **argv) {
7     double A[N], B[N], C[N], D[N];
8
9     for(int i = 0; i < 2; ++i) {
10        cuda-kernel(C, D);
11
12        #pragma omp target device(acc)
13        #pragma omp task in(A) out(B) TASK 2
14        #pragma omp teams distribute parallel for
15        for-loop
16
17        #pragma omp target device(acc)
18        #pragma omp task inout(A, B) TASK 3
19        #pragma omp teams distribute parallel for
20        for-loop
21    }
22    #pragma omp target device(acc)
23    #pragma omp task inout(C, B) in(D) TASK 4
24    #pragma omp teams distribute parallel for
25    for-loop
26
27    #pragma omp target device(smp)
28    #pragma omp task in(A, C) TASK 5
29    structured-block
30
31    #pragma omp taskwait
32 }

```

Figure 2: Simple example code using MACC Infrastructure.

Figure 2 shows an example code using our MACC infrastructure. There are five tasks specified; the device type of 1st task is cuda, the 2nd, 3rd and 4th ones are acc, and the last one is an smp type. In addition, there is a taskwait construct at the end of program to make sure that all the tasks finish before the program is over. For the 1st task, MACC infrastructure seamlessly launches an optimized cuda kernel in line 10. Then, it makes use of the GPU code generation facility for the 2nd, 3rd and 4th tasks. Lastly, for the 5th task it creates a new task on the host device. Our runtime can run this code on a system which has multiple GPU automatically without modifying the code.

1.2 Multi-Target Task Sharing

We propose an extension to the directive-based programming models to support multiple-target task sharing, i.e. the possibility of sharing the execution (of multiple instances) of a task on different devices. We also analyze its implementation in the compiler and runtime system and evaluate its performance in a prototype implementation in the OmpSs programming model. The proposed extension eases the use of multiple accelerators in conjunction with the vector and heavily multi-threaded capabilities in multicore processors without any code

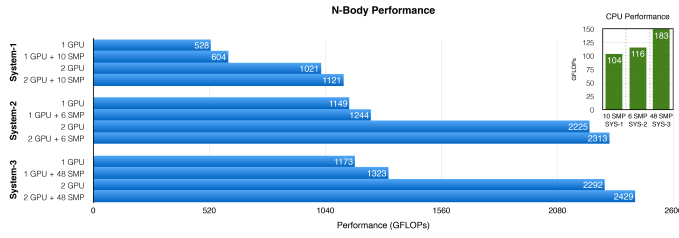


Figure 3: N-Body Simulation Performance Results

modification. The compiler is responsible for the generation of device-specific code for each device kind, delegating to the runtime system the dynamic scheduling of tasks to each available device. The new proposed clause conveys useful insight to guide the scheduler while keeping a clean, abstract and machine independent programmer interface.

Figure 3 shows the performance results obtained for the N-Body kernel on three systems: (1) 2 x Intel 6 core, NVIDIA Tesla M2090, (2) 2 x Intel i7 4 core, 2 x NVIDIA Tesla K40 and (3) 2 x Power8 12 core, 2 x NVIDIA Tesla K40. The performance plot of the top-right corner shows the performance achieved when using the cores in the host for the three first system configurations. The main performance plot in shows how that performance is improved when using one and two GPUs, reaching performance increases in the 8%-14% and 4%-10% ranges, respectively. This contributed performance is very close to the ideal performance which could be obtained by just adding the performance of the CPU to the GPU.

1.3 Lazy Nested Parallelism

In order to give support to more irregular codes (e.g. graph algorithms with irregular data access patterns and unpredictable control flows), the latest releases of GPUs include dynamic parallelism which is able to launch kernels from threads running on the device, without host intervention. Unfortunately, the overhead of launching kernels from the device is higher than launching them from the host. In order to reduce these overheads, this thesis proposes and evaluates a user-directed code transformation technique (LazyNP) that targets nested parallelism. The compiler generates code to dynamically pack kernel invocations and to postpone their execution until a bunch of them are available. Figure 4 shows the overall speed-up on the system which has Intel Haswell 8 core, NVIDIA Tesla K80 relative to original CUDA implementation, OpenMP CPU and the five different flavours of *LazyNP* for the 7 benchmarks. In summary we find *LazyNP Host-based* is the best to implement nested parallelism, which is interesting since it is the only approach that does not require the dynamic parallelism support available in recent GPU devices.

1.4 Dynamic Loop Scheduling

For CPU execution, the number of threads is usually the number of cores, or the number of available thread contexts, which is a small multiple of the number of cores. Two common scheduling decisions are choosing *static* or *dynamic* loop scheduling, and choosing to schedule iterations cyclically or to schedule

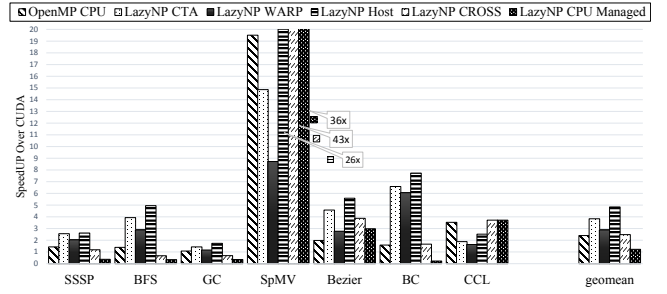


Figure 4: Overall speed-up of LazyNP with respect to baseline original CUDA implementation on Nvidia k80 and Intel CPU

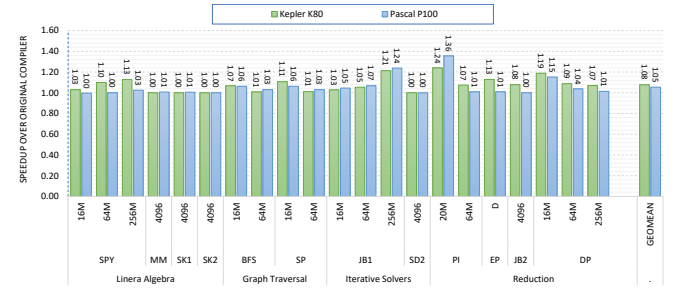


Figure 5: Application based speed-up of optimized dynamic scheduling over static cyclic scheduling.

chunks of consecutive iterations to a thread. GPUs support many more threads within thread blocks of execution, typically thousands of threads. However these threads are short-lived and typically not all threads will be simultaneously resident. When developing or compiling parallel loops for execution on a GPU, the programmer or compiler must again decide how many threads to launch, and which thread will execute each iteration. The simplest mapping is to create one thread for each loop iteration and to statically map the loop iterations sequentially across the threads. However, it is undesirable for directive-based compilers due to the unknown trip count of the parallel loop. For this reason, their compilers generate a kernel with a for-loop which processes the parallel loop of the kernels' threads in a cyclic order. In this way, compilers make sure that the generated kernel completes all the iterations of the parallel loop; also the thread and grid size of the kernel will never exceed the maximum limit. Unfortunately, this method does not yield as good performance as the simplest mapping. To address these issues, we first conduct a thorough exploration of conventional loop scheduling methods on GPUs. We propose the concept of optimized dynamic loop scheduling along with an implementation in the NVIDIA PGI OpenACC compiler. Our method yields better performance than the solution used in directive-based compilers, providing the same performance as the simplest mapping. We conclude with performance comparisons of dynamic scheduling vs. static scheduling by using the latest version of the PGI compiler. Figure 5 shows the speed up of each application.