Facilitating Irregular Applications on Many-core Processors

Da Li
Electrical and Computer Engineering
University of Missouri

1. Introduction

Over the last decade, many-core Graphics Processing Units (GPUs) have been widely used to accelerate a variety of applications. Meanwhile, Intel has released its Xeon Phi Coprocessor, which is equipped with more than fifty x86 cores, each supporting four hardware threads. Despite their widespread use, many-core processors are still considered relatively difficult to program, in that they require the programmer to be familiar with both parallel programming and the hardware features of these devices.

If the effective deployment of regular applications on many-core processors has been extensively investigated, the one of irregular applications is still far from understood. Irregular applications are characterized by irregular and unpredictable memory access patterns, frequent control flow divergence, and a degree of parallelism that is known only at runtime (rather than at compile time). In fact, the amount of parallelism within irregular applications depends on the characteristics of the dataset, rather than solely on its size. Yet, many established and emerging applications are irregular in nature, being based on irregular data structures, such as graphs and trees.

2. Contribution

My research focuses on addressing important issues related to the deployment of irregular computations on many-core processors. Specifically, my contributions are in three directions:

- *Unifying programming interfaces for many-core processors*. We proposed a compilation and runtime framework that generates efficient parallel implementations of generic graph applications for multi-core CPUs, NVIDIA GPUs and Intel Xeon Phi coprocessors. Applications are implemented with a unified, platform-agnostic programming API and then our source-to-source compiler performs platform-specific code transformations and optimizations.
- *Runtime support for efficient execution of applications on irregular datasets*. We analyzed the computational patterns of several irregular applications and found that the dynamic nature of the extracted parallelism makes it impossible to find an optimal solution at compile time. So we proposed a runtime system able to dynamically transition between different implementations with minimal overhead, and investigated heuristic decisions applicable across algorithms and datasets.
- *Compiler support for efficient mapping of applications onto hardware*. We proposed different parallelization templates for efficient code generation across various irregular applications and GPU architectures. In addition, we proposed a compiler-assisted workload consolidation method to enhance the efficiency of kernels with dynamic parallelism on GPUs.

3. Previous Work

3.1. Unifying Programing Interfaces [ICPADS'14]

Although many-core processors are widely used, they are relatively difficult to program, since they require programmers to be familiar both with parallel programming and with the features and the operation of these hardware platforms. This complexity is aggravated by the variety of software stacks used by the various many-core platforms.

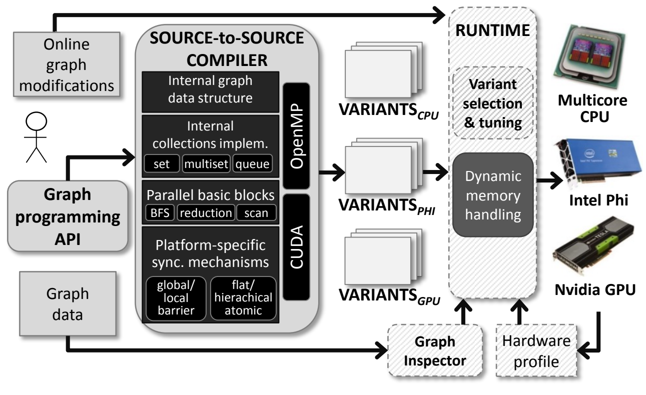


Figure 1. Proposed graph processing system: GRapid.

In this work, we intend to fill this gap and propose a compilation and runtime framework (Figure 1) named *GRapid* for the effective deployment of generic graph applications on many-core processors (GPUs and the Intel Xeon Phi). Our framework also produces multi-threaded code for multi-core CPUs. Unlike previous work, we consider graph applications that use static or dynamic datasets, and we free the programmer from the need to write specific parallel kernels for GPUs and the Intel Xeon Phi. Our framework hides the complexity and heterogeneity of the underlying hardware and software stack from the programmer. The programming API exposed to the user is platform-agnostic, and includes a set of platform-independent sequential and parallel constructs. Our source-to-source compiler converts the graph and the containers (sets, multi-sets and queues) into internal, platform-specific data structures for multi-core CPUs, Intel Xeon Phi and Nvidia GPUs. It then uses iterator-based templates to parallelize graph processing. The compiler generates different functionally-equivalent code variants for the target platforms.

TABLE I: Summary of applications and speedup over serial code.

Application	Application & Graph Type	Attributes	Comp. pattern & arith. intensity	Best Speedup		
				m-CPU	GPU	Phi
BFS	read-only static	Level (int)	Set level (simple& low)	2.5x	50x	3.5x
PageRank	read-only static/dynamic	Rank (double)	Calculate Rank (intermediate & intermediate)	6.3x	6.5x	9x
A-DFA	read-only static	Default Trans (int)	Compare trans. (complex, low)	8x	6x	45x
DFA construction	read-write dynamic	Trans Table (int)	Compare Subset (complex, low)	6.5x	5.5x	4.3x

Table I summarizes the characteristics of the four applications (*BFS*, *A-DFA*, *PageRank*, *DFA Construction*) and the speedup reported on multi- and many-core platforms. In the 2nd column we indicate whether the graph topology is static or dynamic, and, in the latter case, whether it is modified by the application (read-write applications) or by external intervention (read-only applications). The 3rd column shows the application-specific attributes. The 4th column reports an indication of the complexity of the parallel work and its arithmetic intensity. As can be seen, many-core platforms outperform multi-core CPUs on static datasets. The Intel Xeon Phi is preferable to GPU for more complex computational patterns, whereas the arithmetic intensity is not a big discriminating factor. The three platforms report similar speedups on DFA construction; however, the multi-core CPU is in this case a slightly better choice, due to the presence of frequent dynamic memory allocation and synchronization. We note that manually generating code for the three considered devices would be a daunting and time consuming task: by automatically generating different versions of the code, GRapid allows the programmer to quickly identify the platform most suited to his application.

3.2. Adaptive Computing [IPDPS'13]

We analyzed several graphs from the 9th, 10th DIMACS implementation challenges and from the Stanford Large Data Collection and observed that: (i) the graph size varies considerably across different datasets, and (ii) the average node outdegree also varies considerably.

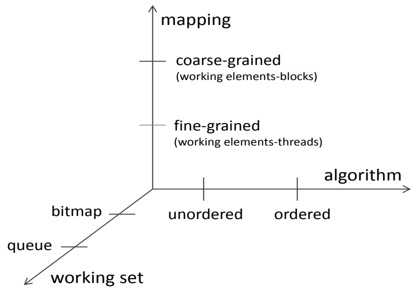
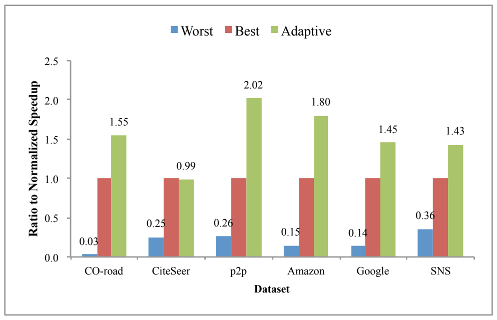


Figure 2: Exploration space.

For graph algorithms on GPUs, we explore an implementation space, which is characterized by the following dimensions: (i) kind of algorithm (i.e., *ordered* vs. *unordered*), (ii) mapping granularity (i.e., *thread-mapping* vs. *block-mapping*), and (iii) working set implementation (i.e., *bitmask* vs. *work queue*). Our analysis shows that there is no optimal static solution across graph problems and datasets. So we argue for an adaptive solution that takes into account the topological characteristics of the dataset to dynamically select the most suitable alternative among a set of available GPU implementations.

Figure 3 shows the speedup comparison between our adaptive runtime, the worst and the best static solution. In every case, the best static implementation is taken as baseline (that is, its speedup value is normalized to 1). Only unordered implementations are considered (also for the worst case). The values reported on the bars are the speedup numbers of the worst static and of the adaptive solution. We can



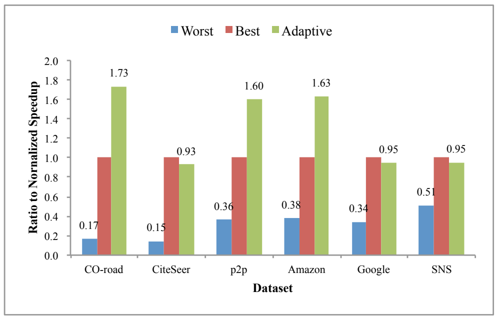


Figure 3: Performance of our adaptive runtime on BFS (to the left) and SSSP (to the right) – baseline: best static solution.

observe the following. First, the performance of the worst static solution can be as low as 3% (and as high as 52%) of that of the best static solution. Second, despite its overhead, our adaptive runtime achieves better performance than the best static implementation on most of these real world graphs. Specifically, the speedup over the best static solution ranges from 1.43 to 2.02. Although on *CiteSeer*, *Google* (*SSSP*) and *SNS* (*SSSP*) the adaptive runtime has no advantage compared to the best static solution, it still achieves 95-99% of its performance (and avoids the penalty associated with possibly choosing a bad implementation).

3.3. Parallelization Templates [GTC'15 and ICPP'15]

The effective deployment of applications exhibiting irregular nested parallelism on GPUs is still an open problem. A naïve mapping of irregular code onto the GPU hardware often leads to resource underutilization and, thereby, limited performance. In this work, we focus on two computational patterns exhibiting nested parallelism: irregular nested loops and parallel recursive computations. In particular, we focus on recursive algorithms operating on trees and graphs. We propose different parallelization templates aimed to increase the GPU utilization of these codes. Specifically, we investigate mechanisms to effectively distribute irregular work to streaming multiprocessors and GPU cores.

Take irregular nested loops in Figure 4(a) as an example. Those templates in Figure 4(b-e) rely on a load balancing threshold parameter (*lb*$_{THRES}$). The *dual-queue* template in Figure 4(b) divides the elements processed in the outer loop in two queues depending on the number of iterations they require in the inner-loop and processes those queues separately using thread- and block-based mapping. The *delayed-buffer* template in Figure 4(c) delays the execution of large iterations of the outer loop by queuing them in a buffer and then processing them using block-based mapping. We consider two versions of this template: one that stores the buffer in global memory (*dbuf-global*); the other that stores the buffer in shared memory (*dbuf-shared*). The *naïve dynamic parallelism* (*dpar-naïve*) template in Figure 4(d) invokes a nested kernel for each "large" iteration (and performs the dynamic parallelism calls at a thread level). Finally, the *optimized dynamic parallelism* (*dpar-opt*) template in Figure 4(e) delays spawning nested kernels to a second-phase; by invoking a single dynamic parallelism kernel for each thread-block, this template spawns fewer and larger kernels.

```
(a) irregular nested loop
foreach (int i = 1 to N)
  foreach (int j = 1 to f[i])
    work(i,j);
(b) dual-queue
set_low = {i :: f[i] ≤ lb_THRES}
set_high = {i :: f[i] > lb_THRES}
thread-mapped_kernel(set_low)
block-mapped_kernel(set_high)
(c) delayed-buffer
thread-mapped-loop(i){
  if(f[i] ≤ lb_THRES)
    for (int j = 1 to f[i]) work(i,j)
  else
    buffer.add(i);
}
blk-mapped-exec(buffer, work);
(d) dynamic parallelism œnaïve
thread-mapped-loop(i){
  if(f[i] ≤ lb_THRES)
    for (int j = 1 to f[i]) work(i,j)
  else
    blk-mapped_nested-kernel_TH(i,work);
}
(e) dynamic parallelism œoptimized
thread-mapped-loop(i){
  if(f[i] ≤ lb_THRES)
    for (int j = 1 to f[i]) work(i,j)
  else
    buffer_SM.add(i);
}
blk-mapped_nested-kernel_BL(buffer_SM, work);
```

Figure 4. Parallelization templates for irregular nested loops.

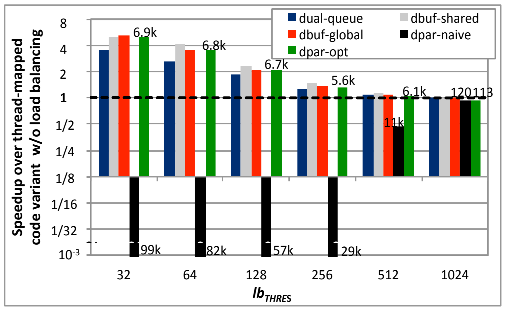


Figure 5. *SSSP*: Speedup of parallelization templates over basic GPU implementation. The numbers on the bars indicate the number of nested kernel calls performed by dynamic parallelism-based solutions.

Figure 5 shows the results reported on an Nvidia K20 GPU (using optimal thread-block configuration). Due to the irregular nature of this graph, almost all templates that include load balancing outperform the basic thread-mapped implementation. Due to the large number of (small) nested kernel calls, however, the naïve dynamic parallelism-based template leads consistently to (often significant) performance degradation. The delayed buffer-based (*dbuf-*) and the optimized dynamic parallelism-based templates yield the best results, and the performance improvement depends on the value of the load balancing threshold (*lb*$_{THRES}$), which affects the amount of load balancing performed. The optimal value of this parameter corresponds to the warp size (no improvements were observed for *lb*$_{THRES}$<32).

4. Ongoing Work [unpublished]

4.1. Workload Consolidation

Dynamic Parallelism (DP) is a hardware feature enabling launching kernels from GPU threads. This feature can be used for various purposes like dynamic load balancing, data-dependent execution and parallelizing recursive algorithms. However, we observed that using DP without paying attention to the overhead often leads to performance degradation. In our previous work [GTC'15, ICPP'15 and COSMIC'15], we discovered that many naïve implementations that use DP lead to a large number of (small) nested kernel launches thus incurring consistent (and often significant) performance degradation.

```
__global__ void parent_kernel() {
   work_item = get_work_item(...)
   prework(work_item)
   if (condition) {
  #pragma dp consldt(block) buffer(default, 256) work(work_item)
      child_kernel<<<block_dim, thread_dim>>(..., work_item, ...)
   } else  work(work_item)
   postwork(work_item)
}
```

(a) Annotated CUDA code (parent kernel)

```
__global__ void parent_kernel() {
   work_item = get_work_item(...);
   prework(work_item)
   if (condition)  insert_buffer(curr)
   else  work(work_item)
   synchronize
   if (thread_id==selected)
      child_kernel_consolidate<<<b_dim_con, t_dim_con>>>()
   synchronize
   postwork(work_item)
}
```

(b) Generated CUDA code (parent kernel)

Figure 6. Pseudo code of parent kernels using our proposed pragma for workload consolidation.

In our current work, we are proposing a software solution to improve the performance of using Dynamic Parallelism on GPUs for irregular applications. We observed that the root cause of performance degradation in codes using DP is the thread-based nested kernel launch model, which often leads to a high overhead of nested kernel invocations. Thus, we believe that consolidating workloads across multiple threads can reduce nested kernel launch overhead dramatically and furthermore, a compiler-assisted approach can decrease the programming complexity and facilitate the use of DP.

Our basic idea is to hold workloads from each thread in a consolidation buffer and defer the handling of these workloads to one or several relatively large child kernels. Based on the software hierarchy of GPUs, we can perform workload consolidation at three granularities: warp-level (consolidating workloads among threads within the same warp), block-level (within the same thread-block) and grid-level (within the whole kernel). These mechanisms have pros and cons in the following aspects: (1) synchronization granularity; (2) load balancing granularity; and (3) memory access patterns. We plan to study these trade-offs in depth.

Figure 6 gives an example of using our proposed pragmas to mark the CUDA source code for workload consolidation. The grammar of the directive is: '***#pragma dp [clause+]***'. As shown in Figure 6(a), the only required modification is the single-line '*#pragma dp*' directive in red, which chooses block-level consolidation. Each buffer can have at most 256 elements and the buffered variable is *work_item*. Figure 6(b) lists the generated parent kernel containing different code sections for buffer insertion, synchronization, and consolidated child kernel launch. Notice that the child kernel is also transformed.

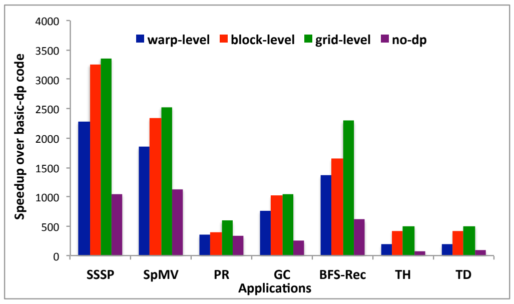


Figure 7. Overall speedup over basic dynamic parallelism.

Figure 7 presents the overall speedup of kernels consolidated at different granularities over *basic-dp* and the results of GPU code without using dynamic parallelism are also included as '*no-dp*'. As can be seen, the *basic-dp* implementation suffers from severe performance degradation due to the significant kernel management overhead and the limited GPU utilization. Even compared with the flat GPU kernels (*no-dp*), *basic-dp* reports slowdown factors from 80 to 1100. On average, warp-level, block-level and grid-level consolidations outperform *no-dp* by a factor of 2.18, 3.26 and 3.78, respectively. The proposed workload consolidation mechanism can significantly improve the performance of GPU kernels with dynamic parallelism.

4.2. Irregular Applications in Distributed Machine Learning

Because of the sparse nature of their datasets and models, many machine learning applications (e.g. probabilistic inference in graphic models, stochastic gradient descents) have irregular computational patterns. Many recent results in machine learning come from two computational breakthroughs: distributed computing and GPU acceleration. We are investigating a combination of compiler and runtime techniques to tackle the following problems: (1) how to fit large models into limited GPU global memory; (2) how to efficiently partition workloads/models between CPUs and GPUs to improve performance; (3) how to schedule distributed machine learning applications on CPU-GPU clusters.