A framework for design space exploration for HPC architectures

Authors Name/s per 1st Affiliation (Author)
line 1 (of Affiliation): dept. name of organization
line 2: name of organization, acronyms acceptable
line 3: City, Country
line 4: Email: name@xyz.com

Authors Name/s per 2nd Affiliation (Author)
line 1 (of Affiliation): dept. name of organization
line 2: name of organization, acronyms acceptable
line 3: City, Country
line 4: Email: name@xyz.com

***Abstract*—In this article we provide a framework for exploiting parallelism onto heterogeneous HPC architectures. Given a HPC cluster with varying compute units, communication constraints and topology, our framework can be utilized for partitioning applications exhibiting task and data parallelism resulting in increased throughput. Our framework can also be used by designers at an early design stage to explore the type of compute units needed and the topology that would be suited for a given application, thereby reducing costs and power requirements. Moreover, software programmers and compiler writers can utilize our framework to gauge the potential parallelism in their programs. The challenge lies in the fact that heterogeneous compute clusters consist of processing elements exhibiting different compute speeds, vector lengths, and communication bandwidths, which all need to be considered when partitioning the application and associated data. We tackle this problem using a staged graph partitioning framework. Our experiments show an order of magnitude speedup for applications. Furthermore our framework finishes within seconds even when simulating 100's of processing elements, which makes our architecture suitable for exploring parallelism potential at compile time.**

***Keywords*-Graph partitioning, vectorization, data parallelism, heterogeneous architectures, clusters.**

```
//Task and data-parallel
for (int i=0;i<M; ++i){
 for (int j=0;j<N; ++j){
  1: A[i][j] = ((((i)*((j)+2.0))+2.0)/(N))
  2: B[i][j] = ((((i)*((j)+3.0))+3.0)/(N))
 }
}
for (int k=0;k<TSTEPS;++k){
 for (int i=1; i<M; ++i)
  for (int j=1; j<N; ++j)
   3: B[i][j] = 0.2*(A[i][j]+A[i][j-1]
              +A[i][j+1]+A[i-1][j])

 //Data-parallel
 for (int i=1; i<M; ++i)
  for (int j=1; j<N; ++j)
   4: A[i][j] = B[i][j]
}
```

Figure 1. Example 2-dimensional Jacobi application

I. Introduction and motivating example

Today's HPC clusters consists of a large number of heterogeneous processing elements such as CPUs, GPUs, DSPs, FPGAs, etc. Given an application exhibiting potential for parallelism the question remains: how does one determine the type of architecture best suited to extract this parallelism? Or given an architecture, how does one determine how to exploit the potential parallelism in the applications.

Consider the code snippet in Figure 1 that carries out the main stencil computation using the Jacobi algorithm. Jacobi is an important stencil computation, which is used for solving large systems of linear equations, especially for heat transfer problems and numerical fluid mechanics. We have chosen this as our motivating example, because there have been attempts to parallelize Jacobi using MPI [**?**] and CUDA [**?**] with success, making it an important problem to solve on a heterogeneous compute cluster mixing MPI and CUDA programming techniques and with the potential of making it faster still. The CUDA programming techniques exploit the *Single Instruction Multiple Data* (SIMD) potential in the Jacobi algorithm [**?**] by modeling parallelism as vector computations suitable for a GPU. The MPI approach on the other hand exploits *Multiple Instruction Multiple Data* (MIMD) potential by modeling the parallelism across different CPU nodes. Both techniques result in 3-4 times speedup compared to single CPU implementations. The challenge when *re-designing* and *tuning* such parallel applications to exploit vector units *or* MPI alone is well documented [**?**], [**?**]. The complexity of re-designing for a mixture of two grows exponentially. The growth in complexity of design space is due to a number of factors, some of which we enumerate below:

- The vector lengths of the underlying processing elements differ. Intel processors have 256 bit vector instructions, while the GPU range varies.
- The actual data-type results in different utilization of vector units. For example, a `double` type requires twice as many vector registers to carry out processing as compared to an `int` type.
- The size of the vector length and the number of vector

units required needs to be determined: simply dividing the data-parallel vector units onto the largest available vector processing elements does not necessarily result in good application throughput or latency. In an ideal scenario for very large vector computations, the vector units can be utilized completely and the rest of the data-parallelism can be exploited in parallel on a CPU unit iteratively in a loop.
- The bottleneck of the communication fabric plays an important role in the partitioning problem. Note that in a heterogeneous compute cluster the communication latencies and bandwidths themselves vary.
- Allocation of data-stores being utilized by the different processing elements needs to be handled.
- Applications written in different ways result in different parallelism potential.
- Finally, the scheduling problem is known to be NP-hard [**?**]. Thus, we need a good heuristic solution, which finishes quickly and gives good results.

There are a number of other applications where parallelism plays and important role. For example, binomial option pricing, k-means calculations, Gauss-Seidel stencil computations, etc, are well suited to be optimized across heterogeneous HPC architectures. In general we have found it is much more essential to exploit data-parallelism as compared to task-parallelism to achieve speedups. But, exploiting both types of parallelism is essential in the general case.

In this paper we propose a framework, which can be used by topology designers, and application writers to quickly carry out a design space exploration to determine the type of underlying topology best suited for a given application. Moreover, compiler writers can also use our framework when vectorizing to partition large vector units onto parallel GPU/CPU units.

II. Related work

A significant amount of research literature exists for extracting parallelism from programs [**?**], [**?**], [**?**], [**?**], [**?**]. The polyhedral optimization model [**?**] concentrates on extracting parallelism from loops, which is a form of data-parallelism. The polyhedral optimization community has concentrated on optimizing for CPUs and GPUs separately, but to our knowledge has not explored the combination of the two. Carpenter et.al [**?**] have again explored ideas for partitioning onto heterogeneous architectures, but at a much smaller scale and again ignoring data-allocation costs and depending upon the polyhedral model for vectorization. The StreamIt [**?**] community has also explored parallelization techniques, but they have only targeted homogeneous RAW [**?**] architecture. There are the classical algorithms such as critical path scheduling [**?**] and list scheduling [**?**], which have been used for scheduling task parallel process onto homogeneous architectures. Declustering [**?**], is another technique, which misses the opportunity to mix SIMD and task-parallel optimizations together.

Cluster based partitioning techniques [**?**], [**?**], [**?**] only consider independent tasks without communication. The proposed heuristics for partitioning data-parallel applications onto clusters [**?**], [**?**] do not consider vectorization potential available on the compute clusters and only concentrate on partitioning task parallel processes.

III. Preliminaries

First of all, in this section, we present a formal description of the problem along with the notations used.

A. Notations

We refer to our application graph, as a *Static Task Graph* (STG) defined formally as: $G_t(V_t, E_t)$, where V_t is the set of all tasks in the application graph and E_t represents the communication between these tasks. The system resources are represented by a weighted undirected graph $G_r(V_r, E_r)$ where V_r represents a set of processing elements which can have different processing capabilities. Each vertex in the task graph, $t_i \in V_t$ is referred to as tasks and each vertex in the resource graph, $r_i \in V_r$ is referred to as processing elements (PEs). We use N_T to denote the total number of tasks in the task graph and N_R to denote the total number of processing elements. By our definition of previous notations, it follows that $N_T = |V_t|$ and $N_R = |V_r|$.

B. Problem Definition

Given a graph $G_t(V_t, E_t)$, each vertex in the task graph, $t_i \in V_t$ has a set of associated requirements represented by T^i_{cj} where $j = 0...n$ with n being the number of requirements for the task graph. These requirements represent the computational requirement of the kernel. Consider the task graph in Figure **??** for the running Jacobi example. The `boxed` statements give the number of instructions and the vector count requirements. The communication edges ($(t_i, t_j) \in E_t$) between data-stores (`boxed` statements without the constraints in Figure **??**) and the execution statements represent the amount of data that needs to be transfered from the store and its utilization at the statement level. The task graphs are generated directly from the program by our compiler. More information about the generation of the task graphs is provided later in Section **??**.

Similarly we have a resource graph $G_r(V_r, E_r)$ where each vertex denotes a processing element. It consists of a set of vertices $V_r = \{r_1, r_2, ..., r_n\}$ and a set of edges $E_r = \{(r_i, r_j) | r_i, r_j \in V_r\}$. Each processing element $r_i \in V_r$ has a set of constraints, that represent the computational capabilities of the processing elements, represented by R^i_{cj} where $j = 0...n$ with n being the number of constratins for the resource graph. For some resource graph as shown in Figure **??**, which has x nodes and 2 constraints. Constraint 1 represents the frequency of the PE or how many