Hetrogeneous Multiconstraint Application Partitioner (HMAP)

Servesh Muralidharan[†], Aravind Vasudevan[†], Avinash Malik[§] and David Gregg[†]
[†]*School of Computer Science and Statistics, Trinity College Dublin, Dublin 2, Ireland*
Email: muralis@scss.tcd.ie, vasudeva@scss.tcd.ie, David.Gregg.cs.tcd.ie
[§]*IBM Research - Ireland*
Email: avinmali@ie.ibm.com

***Abstract*—In this article we propose a novel framework – *Hetrogeneous Multiconstraint Application Partitioner* (HMAP) for exploiting parallelism on heterogeneous *High Performance Computing* (HPC) architectures. Given an hetrogeneous HPC cluster with varying compute units, communication constraints and topology, HMAP framework can be utilized for partitioning applications exhibiting task and data parallelism resulting in increased performance. 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 approach. HMAP framework finishes within seconds even for architectures with 100's of processing elements, which makes our algorithm suitable for exploring parallelism potential.**

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

I. Introduction

High performance computing (HPC) clusters increasingly consist of large numbers of heterogeneous processing elements such as CPUs, graphics processing units (GPUs), field programmable gate arrays (FPGAs), low-power processors intended for digital signal processing (DSP), etc. By combining heterogeneous processing units it may be possible to divide the work so that different types of computation in the application are run on different types of units. This can result in significant speed-ups, lower hardware costs and/or reduced power consumption by the HPC system. For example, if a computation contains the right patterns of data parallelism it may run dozens or even hundreds of times faster on a GPU than on a CPU that has similar cost and power consumption. On the other hand, computations with less data parallelism and more complex control flow may run faster on CPUs. Matching the type of computation to the processor can yield significant benefits. Although the potential of heterogeneous computing is great, exploiting that potential is more difficult.

In this paper we consider the streaming [1] model of computation. Streaming is a popular model for programs such as image and signal processing, financial applications, networking, telecommunications, etc. In the streaming model statements (also called filters/actors/tasks or kernels) execute iteratively, processing the incoming tokens of data. Given such a stream application, it is difficult to map the available parallelism onto the hardware. For example, how does one decide which parallel filters should run on which type of execution unit? Given a system with dozens or hundreds of CPUs, GPUs and other units, how does one divide the work between them? There are several conflicting factors. For example, one wants to allocate filters to the type of execution unit that will execute it most efficiently. On the other hand, one wants to achieve a good load balance by dividing the work evenly across the units. We want to allocate the filters to reduce communication costs while at the same time taking account of all the other factors.

In this paper we consider the problem of mapping graphs of parallel tasks to heterogeneous HPC computing systems. This problem has been studied extensively for homogeneous architectures where all processing elements are the same. Although the homogeneous case is NP-hard [2], several heuristic solutions have been found that work well in practice. However, extending these solutions to the heterogeneous case is difficult for two reasons.

In the heterogeneous case some processing elements are more powerful than others, so achieving a good load balance usually involves distributing the work unevenly.

A second reason why it can be difficult to extend algorithms for homogeneous architectures to the heterogeneous hardware relates to the strengths and weaknesses of different types of processors. When considering heterogeneous architectures, it is tempting to think of some processing elements simply being more powerful than others. A GPU is not simply a more powerful CPU. In fact, some types of computation run better on CPUs and some on GPUs. For a mapping algorithm to work well, it needs to take account of the strengths and weaknesses of different types of processing elements. In this paper we present an approach to mapping parallel tasks to heterogeneous architectures that addresses both of these concerns.

Our **main contributions** are as follows:

- We present a novel approach to characterizing the type of processing elements based on their level of vector parallelism which allows us to distinguish the suitability of different types of units to different filters.
- We provide a novel algorithm for mapping task and data parallelism to heterogeneous architectures based

on hierarchical graph partitioning.
- In addition to mapping filters to processing elements, our framework also allocates the data stores being used by the different filters.

The rest of this paper is organized as follows. We first provide a motivating example in Section II formalizes the problem statement and defines the objective function. Next, in Section III, we provide a detailed description of our framework. Section IV gives the quantitative comparisons of our approach against other approaches. Section V describes the related work and positions our approach in comparison to these works. Finally, we conclude in Section VI.

II. Preliminaries

We now present a formal description of the problem along with the notations used.

A. Execution model

Consider the Jacobi example and its filter graph in Figure 1. The Jacobi algorithm is used in fluid dynamics and heat transfer problems. We consider every statement (marked 1 to 4) in this example to be a filter that can be run in a software pipelined [3] manner on a given architecture. An *example* execution trace of the Jacobi example is shown in Table I for some arbitrary value of computation and communication latency of statements.

P0	1_0	2_0	1_1		2_1	1_2
P1		3_0	4_0	4_0	3_1	4_1

Table I: Example execution trace of the Jacobi kernel

In a software pipelined model, the different iterations of the filters are run in parallel, e.g., 1_0 is the 1^{st} iteration of statement 1 in the Jacobi example, while 1_1 is the second iteration and so on and so forth. The latency of the application (termed makespan) is the period, which iterates continuously (shown within the double lined columns in Table I). In such a model, the resource allocation (rather than dependencies) determines the application latency, especially without back-edges in the filter graph (as is the case with our model). In Table I, the resource allocation on processing element `P1` determines the application latency, because that is the maximum of the two allocation latencies.

B. Notations

We refer to our application graph, as a *Synchronous DataFlow* (SDF) graph defined formally as a weighted directed graph: $G_t(V_t, E_t)$, where V_t is the set of all filters in the application graph and E_t represents the communication buffers between these filters. 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 (PEs) which can have different processing capabilities and E_r represent the communication links between these PEs with differing latencies and bandwidths.

C. Problem Definition

Given a graph $G_t(V_t, E_t)$, each vertex in the filter graph, $t_i \in V_t$ has a set of associated requirements represented by T_j^i where $j = 0...n_t$ with n_t being the number of requirements. These requirements represent the computational requirements of the filter. Namely, T_0^i represents the scalar requirements, while T_1^i represents the vector requirements.

The communication edges are decorated with $e^c \in E_t$, which denotes the data the filter requires for processing.

Each resource node $r_i \in V_r$ has a number of computational capabilities, represented by R_j^i where $j = 0...n_r$. In our formulation $n_t = n_r$. For each node, capability: R_0^i represents the frequency of the PE or how many scalar instructions the PE can perform in one second (the *Million Instructions Per Second* (MIPS) count). Capability : R_1^i denotes the maximum number of parallel vector operations it can perform (the vector length). Each edge $e \in E_r$ has a weight which represents the bandwidth between two PEs r_i and r_j which is denoted by E^c.

The problem at hand is to effectively map the filter graph G_t onto given resource graph G_r. This problem is known to be NP-Hard [2].

Given some filter $t_i \in V_t$ mapped to some resource $r_j \in V_r$, the latency for that node is computed by Equation (1). In this formulation for some filter t_i being mapped onto some resource r_j, we first calculate the number of vectorized instructions that can be executed in parallel (by dividing the required vector length by the vector capacity of r_j represented by R_1^j). We then multiply this number by the number of iterative (non-vectorized) instructions required to get the total number of instructions to be performed by that filter-graph node. Once we have this number we calculate latency of execution of this filter-graph node t_i on this resource r_j by dividing it with the MIPS value of the resource denoted by R_0^j. Calculation of the communication latency requires dividing the number of bits by the bandwidth of the shortest path.

Given the filter-graph and the resource-graph, let $\mathcal{M}$ be some mapping of the application on the resource-graph. For a particular allocation onto some resource node $r_s \in V_r$, we define its computation and communication latency as in Equation (2).

$$((T_1^i/R_1^j \times T_0^i)/R_0^j) + (e^c/E^{c'}) | c = (t_i, t_k), t_k \neq t_i, \forall t_k \in V_t, c' = (r_j, r_l), r_l \neq r_j, \forall r_l \in V_r \tag{1}$$

$$L_s^{\mathcal{M}} = Lcomp_s^{\mathcal{M}} + Lcomm_s^{\mathcal{M}}$$

$$Lcomp_s^{\mathcal{M}} = \sum_{\forall t_i \in V_t} ((T_1^i/R_1^s \times T_0^i)/R_0^s)$$

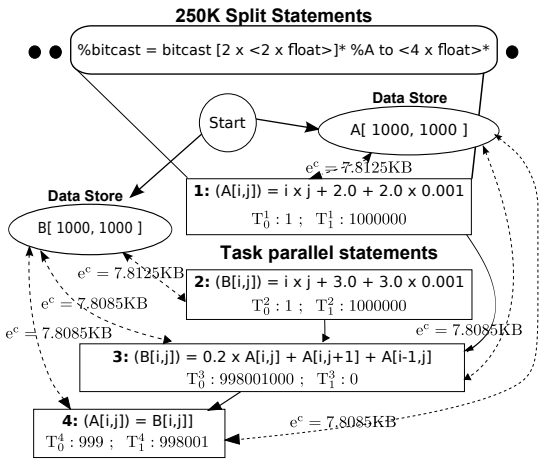
$$Lcomm_s^{\mathcal{M}} = \sum_{\forall t_i \in V_t} e^c/E^{c'}$$

$$s.t., d = (t_i, t_k), t_k \neq t_i, \forall t_k \in V_t \wedge\ c' = (r_s, r_l), r_l \neq r_s, \forall r_l \in V_r \wedge \text{ d is routed on c}' \tag{2}$$

```
//Task and data-parallel
for (int i=0;i<999; ++i){
 for (int j=0;j<999; ++j){
  1: A[i][j] = (i*j+2.0+2.0/1000)
  2: B[i][j] = (i*j+3.0+3.0/1000)
 }
}
for (int k=0;k<1000;++k){
 for (int i=1; i<998; ++i)
  for (int j=1; j<998; ++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<999; ++i)
  for (int j=1; j<999; ++j)
   4: A[i][j] = B[i][j]
}
```

(a) Example 2-dimensional Jacobi application



(b) The filter graph for the Jacobi example [1]

Figure 1: Jacobi example and its filter-graph

Finally, the complete application latency can then be defined as:

$$L^{\mathcal{M}} = max(L_s^{\mathcal{M}}), \forall r_s \in V_r \quad (3)$$

The objective of our framework is to find a mapping $\mathcal{M}$ that minimizes the total application latency as described in Equation (3).

III. HMAP FRAMEWORK

A common approach to solving the homogeneous case is to *partition* the graph across the processing elements [4], [5], [6]. However, we have found that such heuristic partitioning approaches do not work well for heterogeneous architectures. Instead we propose a novel approach where the architecture is hierarchically partitioned into sub-clusters. Each sub-cluster at the same level of the architecture hierarchy has approximately the same computing capability and has relatively local communication. This allows us to use existing approaches that work well for the homogeneous case to partition *heterogeneous* architectures across the sub-clusters at each level of the hierarchy. We show that this is an effective approach if the sub-clusters are well balanced at each level of the hierarchy.

In this section we describe our heuristic framework HMAP. There are two important concepts that need description. First, we describe how the topology clusters are formed from the resource graphs accounting for communication and heterogeneity of the topology. Secondly, given a filter graph with data parallel filters, task parallel filters, and communication extracted as shown in Figure 1(b) how the mapping is performed.

[1]Ellipses represent data stores. Rectangles represent filter nodes. Rounded rectangle represents data parallel nodes. The dots represent other data parallel nodes not shown in the figure. Dashed arrows represent communication between data stores and execution statements. Solid arrows represent dependence edges.

A. Clustering the resource graph

The resource graph represents the cluster of compute nodes on which the filter graph will be executed. A sample resource graph is shown in Figure 2 at level 0. The resource graph that is shown is heterogeneous in both computation and communication. The properties of the resource graph are described below:

- **Compute nodes:** The compute nodes (PEs) are assumed to belong under two categories of processing units, mainly CPUs and GPUs. Following the current trend, the CPUs have a larger MIPS count. The MIPS capacity of a PE is denoted by the first constraint $R_0^i, \forall i \in V_r$. The GPU nodes have a lower MIPS count, but have a large resident vector length, denoted by the second constraint $R_1^i, \forall i \in V_r$. For example, the fifth PE (`E0`) at level 0 is a CPU, since it has a small vector count and a large MIPS count, the first PE (`A0`) on the other hand is a GPU, since the capabilities are reversed.
- **Communication links:** The resource graph shown in Figure 2 at level 0, follows a 2D mesh topology. In this topology the PEs are connected in a grid with individual communication links between them. The bandwidth of these links is non uniform. The different bandwidths on the communication links is represented by the constraint E^C. Our framework can handle any kind of topology, the 2D mesh shown in Figure 2, is just an example topology.

1) Clustering the topology: The main idea behind our partitioning approach is to first hierarchically cluster the nodes in the heterogeneous topology, provided by the designer, thereby forming clusters. The application is then partitioned in stages (levels) onto the resulting hierarchy. The intuition behind this approach is two fold:

- *Heterogeneous K-way partitioning:* The process of partitioning an application onto a given architecture is

equivalent to a heterogeneous K-way partitioning problem. Hierarchically clustering a heterogeneous topology such that the resulting hierarchy consists of clustered PEs with equivalent compute capabilities can reduce the heterogeneous K-way partitioning problem to a homogeneous one.

- *Considering communication links:* The communication can be considered into the equation, while building the hierarchical cluster using the min-cut technique. Thus, a min-cut load-balancing of the PEs in a topology intuitively means: we are clustering together PEs, which have large bandwidth together into a single cluster, while making an attempt to load balance the two capabilities: MIPS and vector lengths.

The hierarchical cluster built for the synthetic topology at level 0 of Figure 2, is shown in the levels 1-3. The stages used to build the hierarchy are as follows:

- *Effective computation during clustering:* Given a topology graph with $|V_r|$ resources, we cluster the PEs in levels, whereby the height of the cluster is $log_2|V_r|$. For example, consider the PEs at level 0 in Figure 2. Clustering from level 0 to level 1 results in 4 PEs at level 1, where the two capabilities, R_0^k, R_1^k for each cluster $k = \{i, j\}, \exists i \in V_r \wedge \exists j \in V_r$ is computed as: $R_0^k = R_0^i + R_0^j$, and $R_1^k = R_1^k + R_1^j$. Without loss of generality we assume this for any $k = \{i, j, .., n\}, \forall n \in V_r$. This process is continued until we reach the top-level with just 1 cluster. We end up with a load-balanced hierarchy, with each level showing a larger amount of homogeneity, and a smaller number of clustered PEs. The reason for a level based clustering, instead of clustering all nodes into a single 2 node partition, is that when partitioning the application on the resulting top-down cluster, we have fine grained details within each of the cluster.
- *Effective communication during clustering:* When clustering nodes the effective communication between two such clusters is hard to determine. This is because the clustering itself is a virtual representation of the actual nodes and between any two nodes across different clusters there might be multiple unique paths. Choosing a suitable path is a routing problem and its beyond the scope of this paper.

  Instead we assume that the link with the least bandwidth would act as the bottleneck in the worst case scenario. These unique links are determined for all the nodes in the cluster. Then to get the *effective bandwidth* between these clusters we aggregate these bandiwdths. The various steps in which this is calculated is explained below:

  1) We use the all-pair Floyd-Warshall [7] algorithm to calculate the shortest path, in terms of latency of data-transfer, for every communication link in the topology. From this we calculate the bandwidth of these links by inverting its latency. This creates a list which contains all of the best case bandwidth between any two nodes.
  2) For any two clusters, we assume a source node in one cluster and determine the paths to all the nodes across the cluster. From this we choose the link with the least bandwidth.
  3) Then we repeat the step above for the other nodes and end up with the paths which would act as a bottleneck for any given unique source and destination pair.
  4) On aggregating the bandiwdths of all these links we end up with the effective bandiwdth between the two clusters.

  For the topology, at level 0, in Figure 2, we first calculate the best possible paths between every pair of nodes $(i, j) \in (V_r \times V_r)$. Now let us consider the example of the clustered nodes `B1` and `C1`. Node `B1` is a cluster of the set of children nodes $\{3, 8, 6\}$ and `C1` is the cluster of the node set $\{2, 1\}$. Being an undirected graph with loss of generality we can consider `B1` to be the source and `C1` to be the destination. Hence, the worst path, in terms of latency of data-transfer, following the all-pair Floyd-Warshall algorithm, between nodes 3 and 2 within the clustered nodes is given by the maximum latency path amongst the memoized best edges: $max((3, 2), (8, 2), (6, 2))$. Similar computation is carried out for all link pairs in the clustered node with 1 as the destination node to find the minimum latency path. Finally, the reciprocal of these two latencies and its addition gives us the effective bandwidth between the two clusters. This effective bandwidth is the overall bottleneck between the two clustered nodes.

B. Task graph partitioning

The filter partitioning on the resulting hierarchical cluster takes a top-down approach. We start with the filter graph extracted from the application (Figure 1). We then recursively partition the filter graph (using K-way partitioning) on the hierarchical cluster of the resource-graph. For the example cluster in Figure 2, we start by partitioning the filter graph in Figure 1, first into two (K=2) partitions considering two equally weighted compute nodes at level 2. Once a partition is obtained for this level, we move onto the next level (level 1), whereby all the filter graph nodes allocated onto node `A2` are further partitioned onto the nodes `A1` and `C1` (again K=2), which are coupled into the cluster `A2`. This process is continued recursively for all clusters until the final level (level 0). During each partition we make sure that the filter node requirements are closely matched to the resource node capabilities.

Doing it in this top down manner has two important consequences.

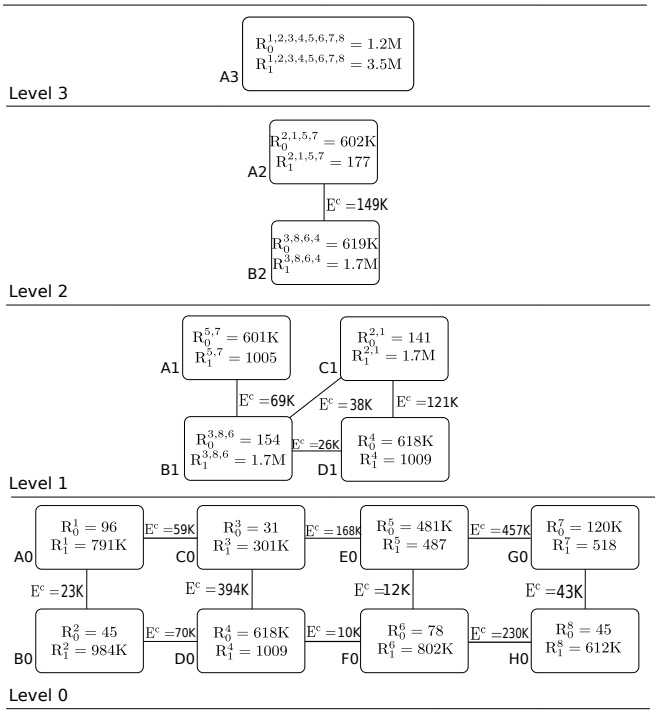


Figure 2: Clustering of a resource graph

- *Increase in the time complexity:* As stated previously, the partitioning problem is equivalent to the K-way partitioning problem. The multi-level K-way partitioning results in the worst cast time complexity of $O(|E_t| \times log_2|V_r|)$ [8]. Our algorithm gives a worst case complexity of $log_2|V_r| \times O(|E_t| \times log_2|V_r|)$ when using the multi-level K-way partitioning. But, in the average case we ask for a balanced 2-way partition at all levels, which results in a complexity of $log_2|V_r| \times O(E_t)$ in the average case.
- *Refined mapping:* By dividing the mapping on to several levels we can achieve much better load balancing by considering only fewer nodes to map than if we were to do it directly on to the resource graph.

IV. Experiments and results

A. Our implementation

We use the Metis [9] graph partitioning library to implement our partitioning algorithm. We are not tied to Metis and any other graph partitioner such as Zoltan [10] or Scotch [11] can be used for implementing our algorithm. Herein, we describe how we used Metis to implement the graph partitioning.

The resource graph is represented in the Metis graph format. We represent the PEs' capabilities as constraints of the nodes and the links' bandwidths as communication weight on the edges. We then construct our clustered structure (Figure 2) by asking for a 2-way partition at each level of the $log_2|V_r|$ height. Metis partitions the graph by load balancing the constraints and performing a minimum edge cut. In partitioning the filter graph, we need to balance the constraints on to the available partitions. Metis offers the ability to load balance multiple constraints on to different partitions based on the metric 'tp-weight'. We calculate the ratios between the capabilities of different partitions and represent them as this metric in order to load balance on to the available partitions.

B. The experimental set-up

The experimental set-up consists of the resource graph generation and the filter graph generation. Herein, we describe the two set-ups.

1) The resource graph set-up: The experimental set up consists of the following.

1) An interconnection network with $|V_r|$ nodes. $|V_r|$ vary from 64 to 4096 PEs. A node can be just a multi-core CPU or a multi-core CPU with an attached GPU.
2) A set of N_G GPUs where N_G is at most $|V_r|$. The GPUs are connected in the network at locations, chosen randomly in the normal distribution of 25% to 75% of $|V_r|$.
3) A set $\boldsymbol{G} = \{G_1, G_2, G_3, ...G_{|\boldsymbol{G}|}\}$ Every GPU in this experiment has a vector length of G_i where G_i is sampled randomly from the set $\boldsymbol{G}$. The elements of set $\boldsymbol{G}$ are chosen from a normal distribution ranging from: 10000 to 100000.
4) A set $\boldsymbol{C} = \{C_1, C_2, C_3, ...C_{|\boldsymbol{C}|}\}$ Every CPU in this experiment has C_i cores where C_i is sampled randomly from the set $\boldsymbol{C}$.
5) A set $\boldsymbol{M} = \{M_1, M_2, M_3, ...M_{|\boldsymbol{M}|}\}$ Every $C_i \in \boldsymbol{C}$ and GPU in this experiment has a MIPS count of M_i where M_i is sampled randomly from the set $\boldsymbol{M}$. The elements of set $\boldsymbol{M}$ are chosen from a normal distribution ranging from: 1000 to 100000.
6) A set $\boldsymbol{B} = \{B_1, B_2, B_3, ...B_{|B|}\}$ Every $|E_r|$ node in this experiment has a bandwidth of B_i in MB/s where B_i is sampled randomly from a normal distribution ranging from: 100 to 100000

For given values of $|V_r|$, N_G, $\boldsymbol{G}$, $\boldsymbol{C}$, $\boldsymbol{M}$ and $\boldsymbol{B}$ and a given application, let the k-th trial be defined as one execution of the following sequence of steps.

- For each GPU G_i, sample $\boldsymbol{G}$ and $\boldsymbol{M}$ randomly to determine its vector length V_i and MIPS count M_i.
- For each CPU P_i, sample $\boldsymbol{C}$ randomly to determine the number of cores C_i in the processor P_i.
- For each core C_i in the processor P_i sample V_i and M_i randomly from set $\boldsymbol{G}$ and $\boldsymbol{M}$.
- Use our framework to extract data and filter parallelism that is best utilizable by the heterogeneity created by parameters in items 1, 2, and 3 above. Determine the execution time $L^{\mathcal{M}}$.

An experiment, $\boldsymbol{E}$ ($|V_r|$, N_G, $\boldsymbol{G}$, $\boldsymbol{C}$ $\boldsymbol{M}$, $\boldsymbol{B}$), consists of conducting enough of the above trials so that width of the

95% confidence interval on the average value of $Latency^{\zeta,\mathcal{M}}$ is less than 10% of the average value. This results in a variable number of trials with different experimental set-ups. Note that two trials differ from each other only in the seed for the random number generator. This reduces the dependence of our results on a lucky sequence of numbers from the random number generator.

2) Random filter graph generation: We built a random graph generator to test our partitioning methodology rigorously. The random graph generator needs as input the following parameters:

- Number of nodes (n) - Total number of nodes to be present in the filter graph
- Indegree (i) - Average indegree of every vertex
- Outdegree (o) - Average outdegree of every vertex
- Communication to Computation Ratio - CCR (c) - It is the ratio of the average communication cost of an outedge the average computation cost of the vertex itself. If a DAG's c is low, then it can be called a computation intensive application and if it is greater than 1 it can be called a communication intensive application
- Structure of the graph (α) - We generate the height of the graph based on α as $\frac{\sqrt{n}}{\alpha}$. This implies the width of the graph becomes $\sqrt{n} * \alpha$. Higher values of α give wider graphs which means the graph has more inherent task parallelism, while lower values give taller graphs which means the graph is inherently serial
- Beta (β) - We use this parameter to decide if an actor in the filter graph is CPU intensive or GPU intensive. Smaller values of β makes actors CPU intensive (by making first constraint larger than the second), while larger values make it GPU intensive (by making the second constraint larger than the first)
- Skewedness factor (γ) - This parameter dictates how computation is spread across the graph. Smaller values of γ give uniformly distributed values for the constraints of the actors while larger values produces skewed graphs

For our experiments we generated random graphs by choosing values for the input parameters from the following sets :

- $\chi_n = \{128, 256, 512, 1024, 4096, 8192, 16384\}$
- $\chi_o = \{2, 4, 8\}$
- $\chi_c = \{0.0001, 0.001, 0.01, 0.1, 1\}$
- $\chi_\alpha = \{0.1, 1.0, 10.0\}$
- $\chi_\beta = \{5, 25, 50, 75, 95\}$
- $\chi_\gamma = \{5, 25, 50, 75, 95\}$

We generated one graph per combination for a total of 7875 application graphs. Since the random graph generator has a variety of inputs and these inputs are filled in from a large set of possible values, a diverse set of DAGs are generated with various characteristics. Experiments based on diverse set of DAGs prevent biasing towards a particular partitioning algorithm.

C. Experimental Results

The experimental set-up consists of a dual socket system consisting of Intel Xeon E5620 CPU running at 2.4Ghz with 24GB DDR3 RAM. The system is running Linux kernel ver 3.0.40-1. Our framework was compiled using gcc version 4.7.1 with '-O3' optimization flag.

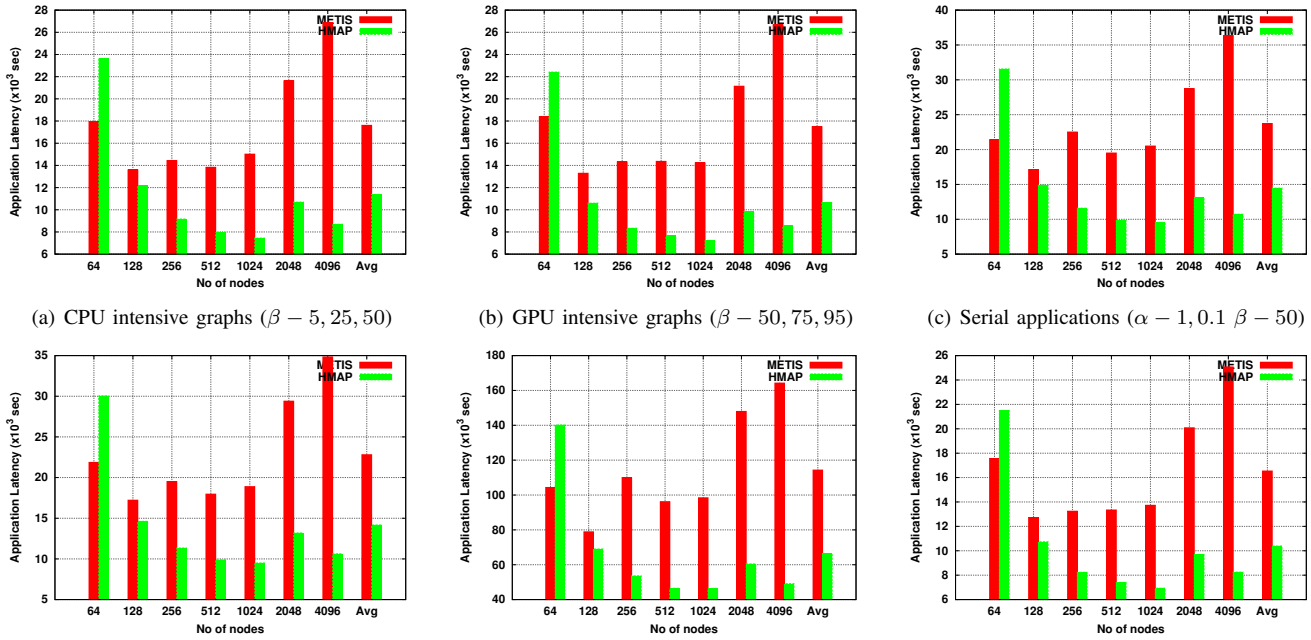
A total of 55120 experiments were performed on each case individually. The results comparing K-way partitioning using metis and HMAP framework are shown in 3. The results are divided into a total of 5 graphs each representing a common characteristic of the application and finally 3(f) shows the average application latency based on all the input graphs. Overall it just took us 50 seconds more than metis for the biggest architecture of 4096 nodes to determine a partition.

Figure 3(a) and 3(b) represents application graphs that consists of proportionally larger CPU and GPU intensive filters respectively. On all archtitectures we perform consistently better and on an average we achieve better results in the case of GPU intesive applications, whereas metis which uses k-way partitioning alone provides almost similar latencies.

In the case of figure 3(c) and 3(d) which represents applications that are tall consisting of more filters that can only be executed serially and fat consisting of more task parallel filters, the trend continues and we are able to provide better latencies in the case of larger archtiectures when compared with metis alone. We attribute this to the following, for smaller architectures, the constraints(tp-weights) are enough to show the difference between machines to metis. Whereas, when the architecture becomes big, the relative difference between two partitions become so small that metis is unable to tell them apart. The problem becomes one of unexpressability. Our framework alleviates this by asking for a very small number of partitions(usually in the range of 2 to 4) at every stage.

The figure 3(e) which shows the set of application graphs which are highly communication intensive determined by the CCR $c = 1$, we perform significantly better in comparison to metis for larger archtiectures. This is due to our clustering approach that ensures topology nodes that communicate with high bandwidths are combined together at each level. Also, metis is unaware of the communication between nodes in the topology as it only cares about a min-cut across partitions. Our framework on the other hand, takes into account the communication in the topology and indirectly matches heavy edges from the filter graph onto high bandwidth edges in the topology graph.

We also observed that metis performs better than our framework in the case of rectangular topologies. We believe this is because rectangular topologies are harder to collate



(a) CPU intensive graphs ($\beta - 5, 25, 50$)

(b) GPU intensive graphs ($\beta - 50, 75, 95$)

(c) Serial applications ($\alpha - 1, 0.1\ \beta - 50$)

(d) Task parallel applications ($\alpha - 10, 1\ \beta - 50$)

(e) Comm. intensive graphs ($\alpha - 1\ \beta - 50\ c - 1$)

(f) Overall

Figure 3: Experimental results based on random application graphs.
Each graph shows the latency for a given architecture and application graphs that belong to a certain criteria

into homogenous clusters as there are straddlers in non-square architectures.

Finally, in figure 3(f) we see that we are able to scale better when compared with metis. In smaller architecture sizes metis is able to produce good mapping, whereas on larger architecture sizes we are able to perform over several magnitudes better in comparison.

V. Related work

A significant amount of research literature exists for extracting parallelism from programs [12], [13], [14], [15], [16]. The polyhedral optimization model [12] 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 [16] 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 [17] community has also explored parallelization techniques, but they have only targeted homogeneous RAW [18] architecture. There are the classical algorithms such as critical path scheduling [19] and list scheduling [20], which have been used for scheduling task parallel process onto homogeneous architectures. The list scheduling techniques targeting heterogeneous architectures such as [21] do not exploit SIMD parallelism onto vector processors. Declustering [15], is another technique, which misses the opportunity to mix SIMD and task-parallel optimizations together.

Cluster based partitioning techniques [22], [23], [24] only consider independent tasks without communication. The proposed heuristics for partitioning data-parallel applications onto clusters [25], [26] do not consider vectorization potential available on the compute clusters and only concentrate on partitioning task parallel processes.

Random search techniques use a specific objective and random choices to guide them through a search space to arrive at a solution. The objective is usually derived from information obtained from previous random searches or prior knowledge about the system. Genetic Algorithms (GA) [27], [28], [29], [30] is a commonly used algorithm from this technique and generate reasonably good results. The drawback being this requires significantly more time than heusristic based algorithms [24]. Moreover these techniques often give such good results due to a unique set of parameters. Determining such optimal parameters to fine tune these random techniques to get good results is a challenge of its own. Moreover Parameters that give good results for a certain scenario may not be the same for a different one.

Several other techniques also exist that belong to the same category in addition to GA such as simulated annealing [28], [31] and local search methods [32], [33] which suffer from the same disadvantages.

VI. Conclusion

In this paper we have described a novel staged graph based partitioning technique to partition and schedule appli-

cations onto heterogeneous execution architectures. HMAP framework considers both filter and data parallelism, which allows the application writers to design and tune their applications to extract parallelism. We also consider communication along application and underlying architecture edges, which when combined with the filter and data parallelism gives a better estimate of the application latency than the currently described research literature targeting similar problems.

We have tested our framework on a statistical sample of randomly generated filter graphs with varying compute, vector, communication requirements.

VII. Acknowledgment

This work is partly funded by the IRCSET Enterprise Partnership Scheme in collaboration with IBM Research, Ireland.

References

[1] J. Buck and E. Lee, *The Token Flow Model*. Advanced Topics in Dataflow Computing and Multithreading, Wiley IEEE Computer Society, 1995.

[2] V. Sarkar, "Partitioning and scheduling parallel algorithms for execution on multiprocessors," Ph.D. dissertation, Stanford university, 1989.

[3] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil, "Software Pipelined Execution of Stream Programs on GPUs," in *CGO*, 2009, pp. 200–209.

[4] A. Aletà, J. M. Codina, J. Sánchez, and A. González, "Graph-partitioning based instruction scheduling for clustered processors," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 34. Washington, DC, USA: IEEE Computer Society, 2001, pp. 150–159.

[5] K. Purna and D. Bhatia, "Temporal partitioning and scheduling data flow graphs for reconfigurable computers," *Computers, IEEE Transactions on*, vol. 48, no. 6, pp. 579 –590, jun 1999.

[6] E. Nystrom and A. E. Eichenberger, "Effective cluster assignment for modulo scheduling," in *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 31. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998, pp. 103–114.

[7] S. S. Skiena, *The Algorithms Design Manual*, 2nd ed. Springer-Verlag London, 2008.

[8] G. Karypis, V. Kumar, and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed Computing*, vol. 48, pp. 96–129, 1998.

[9] G. Karypis and V. Kumar, "Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0," Tech. Rep., 1995.

[10] K. Devine, E. Boman, L. Riesen, U. Catalyurek, and C. Chevalier, "Getting started with zoltan: A short tutorial," in *Proc. of 2009 Dagstuhl Seminar on Combinatorial Scientific Computing*, 2009, also available as Sandia National Labs Tech Report SAND2009-0578C.

[11] C. Chevalier and F. Pellegrini, "Pt-scotch: A tool for efficient parallel graph ordering," *Parallel Comput.*, vol. 34, no. 6-8, pp. 318–331, Jul. 2008.

[12] M. Griebl, C. Lengauer, and S. Wetzel, "Code Generation in the Polytope Model," in *In IEEE PACT*. IEEE Computer Society Press, 1998, pp. 106–111.

[13] J. Donald and M. Martonosi, "An Efficient, Practical Parallelization Methodology for Multicore Architecture Simulation," *IEEE Computer. Architecture. Letter.*, vol. 5, pp. 14–, July 2006.

[14] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," *SIGOPS Oper. Syst. Rev.*, vol. 40, pp. 151–162, October 2006.

[15] G. C. Sih and E. A. Lee, "Declustering: A new multiprocessor scheduling technique," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, pp. 625–637, June 1993.

[16] P. M. Carpenter, A. Ramirez, and E. Ayguade, "Mapping stream programs onto heterogeneous multiprocessor systems," in *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, ser. CASES '09. New York, NY, USA: ACM, 2009, pp. 57–66.

[17] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *CC '02: 11th International Conference on Compiler Construction*, London, UK, 2002, pp. 179–196.

[18] E. Waingold, M. Taylor, V. Sarkar, V. Lee, W. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktumi, R. Barua, J. Babb, S. Amarsinghe, and A. Agarwal, "Baring it all to software: The raw machine," MIT, Cambridge, MA, USA, Tech. Rep., 1997.

[19] W. Kohler, "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems," *Computers, IEEE Transactions on*, vol. C-24, no. 12, pp. 1235 – 1238, dec. 1975.

[20] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Commun. ACM*, vol. 17, no. 12, pp. 685–690, Dec. 1974.

[21] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 3, pp. 260 –274, mar 2002.

[22] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic mapping of a class of independent tasks onto heterogeneous computing systems," *J. Parallel Distrib. Comput.*, vol. 59, no. 2, pp. 107–131, Nov. 1999.

[23] A. Doğan and F. özgüner, "Genetic algorithm based scheduling of meta-tasks with stochastic execution times in heterogeneous computing systemst1," *Cluster Computing*, vol. 7, no. 2, pp. 177–190, Apr. 2004.

[24] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *J. Parallel Distrib. Comput.*, vol. 61, no. 6, pp. 810–837, Jun. 2001.

[25] S. Sanyal and S. Das, "Match : Mapping data-parallel tasks on a heterogeneous computing platform using the cross-entropy heuristic," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, april 2005, p. 64b.

[26] S. Kumar, S. K. Das, and R. Biswas, "Graph partitioning for parallel applications in heterogeneous grid environments," in

Proceedings of the 16th International Parallel and Distributed Processing Symposium, ser. IPDPS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 167–.

[27] E. S. H. Hou, N. Ansari, and H. Ren, "A genetic algorithm for multiprocessor scheduling," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 5, no. 2, pp. 113–120, Feb.

[28] P. Shroff, D. W. Watson, N. S. Flann, and R. F. Freund, "Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environments."

[29] H. K. Singh and A. Youssef, "Mapping and scheduling heterogeneous task graphs using genetic algorithms," Master's thesis.

[30] L. Wang, H. J. Siegel, and V. P. Roychowdhury, "A genetic-algorithm-based approach for task matching and scheduling in heterogeneous computing environments," in *Proc. Heterogeneous Computing Workshop*, 1996.

[31] L. Tao, B. Narahari, and Y. C. Zhao, "Heuristics for mapping parallel computations to parallel architectures," in *Heterogeneous Processing, 1993. WHP 93. Proceedings. Workshop on*, Apr, pp. 36–41.

[32] M.-Y. Wu, W. Shu, and J. Gu, "Local search for dag scheduling and task assignment," in *Parallel Processing, 1997., Proceedings of the 1997 International Conference on*, Aug, pp. 174–180.

[33] Y.-K. Kwok, I. Ahmad, and J. Gu, "Fast: a low-complexity algorithm for efficient scheduling of dags on parallel processors," in *Parallel Processing, 1996. Vol.3. Software., Proceedings of the 1996 International Conference on*, vol. 2, Aug, pp. 150–157 vol.2.