The Thesis Committee for Cagri Eryilmaz
certifies that this is the approved version of the following thesis:

Fine-Grain Acceleration of Graph Algorithms on a Heterogeneous Chip

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor: _____

Mattan Erez

_____

Keshav Pingali

Fine-Grain Acceleration of Graph Algorithms on a Heterogeneous Chip

by

Cagri Eryilmaz, B.S.

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2017

Dedication

To my late dad,

and to my beloved mother and sister.

Acknowledgements

I would like to thank Mattan Erez for his guidance and mentorship throughout my three years at ECE department. His great vision and knowledge on computer architecture introduced me the needed insight in graduate school. Working with him and LPH team was a great chance.

I'd also like to thank all ECE Architecture, Computer Systems, and Embedded Systems (ACSES), formerly known as Computer Architecture and Embedded Processors, track professors. The courses they taught and all the conversations I had with them made me a better future computer architect.

I'd also like to thank Keshav Pingali. He and his team provided great feedback during the evaluations and writing of this thesis.

I'd also like to the people I worked with at AMD Research and Microsoft Research. Especially I would like to thank Oguz Ergin, Osman Unsal and Adrian Cristal for their great support for my development.

This work, without my family's limitless support would not be achievable.

Abstract

Fine-Grain Acceleration of Graph Algorithms on a Heterogeneous Chip

Cagri Eryilmaz, MSE

The University of Texas at Austin, 2017

Supervisor: Mattan Erez

With the rise of heterogeneous chips available in the market, where integrated GPU cores and CPU cores reside in the same chip and share a unified memory, it is possible to have better execution schemes for many graph algorithms. Graph algorithms can exhibit producer-consumer behavior, a varying amount of parallelism during execution, and irregularity which results in inefficiency. The inefficiency problem could be solved by exploiting heterogeneity between cores. In this work, I provide an understanding of the executions of some graph algorithms in heterogeneous chips and accelerate their executions by using fine-grain software optimization techniques. To achieve this, I introduce two different fine-grain execution techniques to accelerate the Maximal Independent Set and Preflow-push graph algorithms, and present an evaluation of the techniques on a heterogeneous chip. My techniques, namely Overlapping Threads with Hot-Vertices and Task Switcher, provide 1.3x to 16x speedup over CPU-only execution depending on the input and the algorithm.

Table of Contents

List of Figures

I. Introduction

I.I Thesis Statement

Accelerating irregular applications on a commonly available heterogeneous processor with fine-grain software optimization techniques yields promising speed-up.

I.II Contributions

I provide a compile time solution which achieves significant speedup for a given graph algorithm and sparse inputs on a heterogeneous chip where CPU and GPU cores are incorporated in a single die sharing unified memory space. The graph algorithms I have chosen, namely Maximal Independent Set and Preflow-Push, cover producer-consumer behavior, a varying amount of parallelism, and complex kernels which can be seen in any parallel application. The techniques I develop, namely Overlapping Threads, Hot-Vertices and Task Switcher provide 1.3x to 16x speedup depending on the algorithm and input. The Overlapping Threads technique benefits from fine-grain functions of OpenCL 2.0 allowing different kernel threads to overlap such that total execution time is reduced. Hot-Vertices provide a technique to lessen memory pressure by finding the vertices with highest degrees. Task Switcher allows different task assignments to be made between CPU and GPU cores during a parallel execution.

I.III Organization

At the following background section, hardware and software changes over time, why fine-grain computation is important and why heterogeneous chips are promising are

discussed. Then, in Section III, I discuss my motivation. The details of the framework are discussed in section IV. The methodology and evaluations are provided in Section V and VI respectively.

II. Background

II.I Heterogeneity in Hardware

The idea of heterogeneous cores in a chip has been widely anticipated and studied. The first ideas of heterogeneous cores in a chip started with the small core-big core idea where the big cores accelerated the critical sections of an algorithm while smaller and energy-efficient cores [2] [3] executed the non-critical sections of an algorithm.

Later, as large-scale integration techniques improved and realizing single-instruction multiple-data (SIMD) units are better suited to parallel applications requiring high-throughput, the computer architects started to analyze the benefits of combining CPU and GPU cores in the same die to satisfy the energy efficiency and throughput need of such parallel applications. Examples include AMD Fusion or following APUs such as Kaveri [4] or Carrizo [5] and Intel Sandy-Bridge [6] and following products by Intel. Intel's Sandy Bridge is the first commercial processor that integrates GPUs on a chip with CPU cores, and AMD's Fusion architecture (later named and incorporated as Heterogeneous System Architecture (HSA)) integrates more powerful GPUs on the same die [7] than the former.

Heterogeneous chips where GPU cores and CPU cores embodied together have an important feature which helps programmers and architects to achieve better execution schemes: sharing the same physical memory where GPU and CPU cores can communicate without copying data and without crossing a slow peripheral bus. This in return results in better coherency between the cores as well as provides a better underlying system for development of fine-grain software techniques.

II.II SOFTWARE

While such developments are happening in hardware, computer scientists in the big data field working on complicated problems such as complex networks, knowledge discovery, pattern recognition, and language understanding are trying to tackle the significant problem of irregularity in their applications [10] [11]. Irregularity in parallel applications causes inefficient executions on high-performance systems, such as discrete GPU, which is designed for executing high locality and regular, partitionable structures. The workload irregularity which is caused by the scarcity of the input is characterized by irregular control flow, irregular communication patterns, and irregular data structures [12]. Pingali et al. [31] provide a system that automatically tackles the problems of irregularity of a parallel application with significant compiler level techniques, however they do not explore fine-grain interaction between different types of cores.

As Feo et al. [12] note, the research shifted toward providing optimization techniques for the software or optimizing the hardware by integrating specialized cores, in my case, GPU cores. Heterogeneous chips can be a candidate for exploring such fine-grain execution methods. It is because heterogeneous chips support shared virtual memory and coherency between the different types of cores which would enable us to explore fine-grain execution methods. Nevertheless, by the nature of heterogeneous chips, they also exhibit some problems. To name a few, task scheduling [15] and utilization of the memory system [16] [17] are two important areas of research for heterogeneous chips. Research in these areas is still ongoing, Panneerselvam et al. [15] provide a dynamic solution (runtime) to task scheduling, which aims to achieve high utilization of available cores. Jeong et al. [38] provide a mechanism to dynamically adjust the Quality-of-Service policy of memory

controllers so that performance would not be bottlenecked by inefficient usage of memory. Wang et al. [16] offer a memory request scheduling policy for heterogeneous chips, resulting in an improvement in row buffer locality, service latency of CPU memory requests and throughput of the overall system. Ausavarungnirun et al. [39] provides a staged memory controller for integrated CPU-GPU systems, where memory requests are categorized and passed to three different stages in the memory controller.

II.II.I Producer-Consumer Irregular Applications

Some irregular applications show producer-consumer behavior. Maximal Independent Set (MIS) [21] algorithm can be given as an example to this behavior. Although the details are provided in later sections, in such algorithms, one of the kernels is producing data, often in a linear fashion, such as generating a random number or making it available depending on a condition, and then consumer kernel uses the data. It is important to focus on producer-consumer applications in finer-grain execution since their collaborative nature can help us to develop better insight and techniques in finer grain. When combined with heterogeneous architectures, algorithms like MIS, can benefit from fine-grain software optimization techniques implemented for such architectures. The benefits can include less idle-time for available cores in a heterogeneous die during execution, better utilization of the memory and efficient task placement decisions, for which I provide software techniques in fine-grain in this work.

II.III Fine-Grain Computation

Fine-grain execution techniques are already studied for the high-performance computing systems. The studies such as I-Structures [23] and J-Machine multicomputer

[24] provide a better understanding of data structures suited for efficiency, parallelism, and ease of coding or communication methods in the fine-grain world.

Recently, Kim et al. [25] developed a fine-grain work list technique for GPGPUs targeting irregular applications. The technique implements fine-grain work-lists in HW to improve the algorithmic efficiency of data-driven irregular applications. Ributzka et al. [26] studies the feasibility, usefulness, and trade-offs of fine-grain in-memory synchronization support in a real-world large-scale many-core chip, IBM Cyclops-64, inspired from the Cray XMT's [13] fine-grain in memory synchronization. Grossman et al. [27] introduces a hardware block to Anton 2, a massively parallel special purpose supercomputer for molecular dynamics simulations, which provides flexible and efficient support for fine-grained event-driven computation. Another fine-grain multithreading technique [28] aiming energy efficiency has been studied for in-order and out-of-order processors. Observing that at any instance during the parallel execution only a small fraction of memory locations are actively participating in synchronization, [29] proposes a fine-grain synchronization buffer for multiple threads in a multicore system.

III. Motivation

Heterogeneous architectures where GPU cores and CPU cores are combined within a single processor have an important feature which helps programmers to achieve better execution schemes: unified memory system. Allowing CPU and GPU cores to share the virtual memory space via common page tables, address translations, and process space identifiers [8] enables the implementation of fine-grain execution techniques. OpenCL 2.0 [18] version provides fine-grain shared virtual memory functionality allowing any available devices on any system communicate via shared space, as long as there is support from hardware [19]. Nevertheless, the possibilities that can be achieved with heterogeneous chips with such coding platforms are yet to be explored.

Among the irregular algorithms, producer-consumer behavior is commonly seen, such as in graph coloring [20] and maximal independent set [21]. In such algorithms, in addition to problems of irregularity, the programmers has to come up with solutions for data dependency between producer and consumer kernels in order to lessen idle times on the cores. The idea of heterogeneity can be a solution for this problem as well.

In this work, my contributions are as follows:

1. Provide a framework achieving significant speedup in a given heterogeneous chip for a given irregular graph application. Two examples include Maximal Independent Set for producer-consumer behavior and Preflow-Push for varying amount of parallelism with more complex kernels.

2. Explore a technique for producer-consumer graph algorithms which aims to overlap production and consumption, called Overlapping Threads

3. Develop a basic compile-time task assigner and a model associated with it which finds switching point in a parallel application.

4. Explore a technique for reducing the memory pressure caused by large amount of polling due to SW fine-grain implementation, called Hot-Vertices.

IV. Fine-Grain Acceleration of Graph Algorithms on Heterogeneous Chips

To accelerate the algorithms with fine grain optimization techniques, I propose two main techniques in this compile time framework, as can be seen in Fig 1. The first one, named *Overlapping Threads,* overlaps producer and consumer threads in a given producer-consumer application. To tackle the problem of increased memory pressure, I propose the *Hot-Vertices* technique, combined with *Overlapping Threads*. The second technique, named the Task Switcher, provides an execution model and task placement decision maker before execution. The detailed discussion of each technique is given in the following sections.

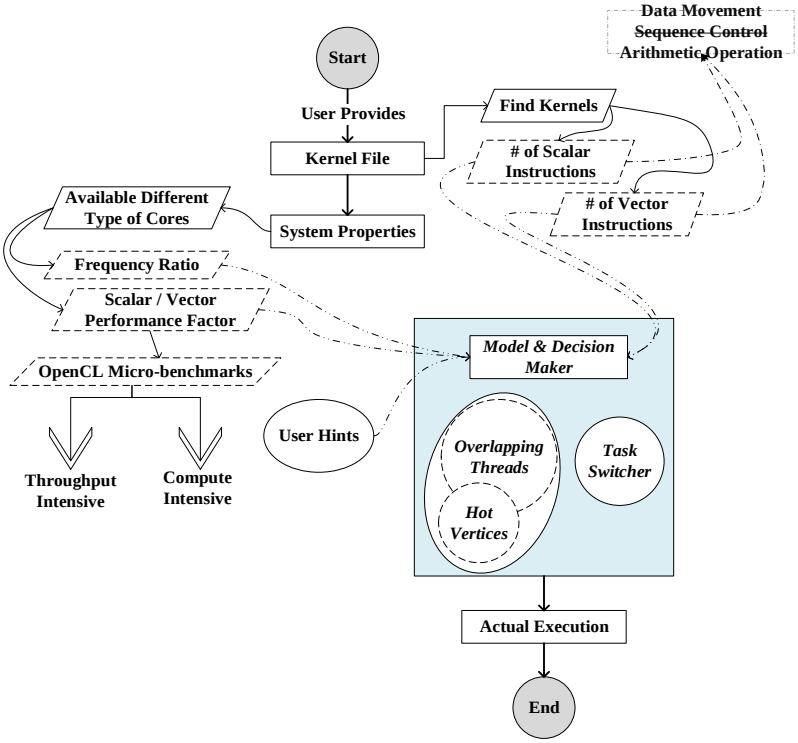

Figure 1: The flow of the framework

IV.I OVERLAPPING THREADS TECHNIQUE

In any given producer-consumer irregular application we have producer kernel(s) and consumer kernel(s). Normally, these kernels would be executed in a serial fashion, that is after the producer kernel completes its execution, consumer threads would be able to launch and consume the producer data as shown in Figure 2. I take this execution as the baseline execution model for comparison.

In Overlapping Threads, I try to benefit from the idea that some of the consumer threads could actually execute without even waiting until the producer kernel completes. To achieve this I utilize OpenCL 2.0 [30] Shared Virtual Memory (SVM) functions, and I implement *a fine-grain full-empty bit structure* in software. This structure is used as a fine grain synchronization point for different kernels. For example, for producer-consumer applications, fine-grain full-empty bit software structure allows consumer threads to check whether the data is produced or not, which is done via polling on the structure. This structure is an array of bits, where each bit denotes whether the corresponding data has been produced or not. The length of the array is equal to number of items or nodes to be processed. As a general rule of thumb in OpenCL 2.0 SVM fine-grain implementations, the writes and reads to any shared memory address is done via SVM atomic functions.

I study varying amounts of overlap focusing on the best execution time and energy consumption. The overlap amount ranges from 20% to 99%. The amount defines when to launch consumer kernels. For example, 20% overlap states, after producing 80% of the data, the consumer threads are launched, that is, while producing the *last* 20% of the data, both producer and consumer threads are overlapped. The technique is illustrated in Fig 3. The diagram on the left shows 20% overlap where at every computational step while

producing the last 20% of the data, the consumer kernels are launched. The diagram on the left is for 99% overlap.

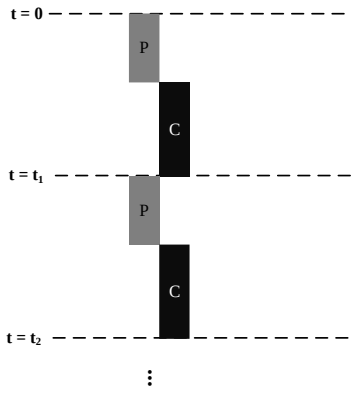

Figure 2: Baseline producer-consumer model

With OpenCL 2.0 [30] it is possible to use shared virtual memory (SVM) functions. With SVM, the same address space is exposed to both the host and the devices within a given context. This allows the programmer to use pointer-based data structures in OpenCL such as graphs or lists. The OpenCL manual specifies three types of SVM:

1. Coarse-Grained buffer SVM: Sharing occurs at the granularity of regions of memory objects
2. Fine-Grained buffer SVM: Sharing occurs at the granularity of individual loads/stores in memory objects.
3. Fine-Grained system SVM: Sharing occurs at the granularity of individual loads/stores occurring anywhere within the host memory.

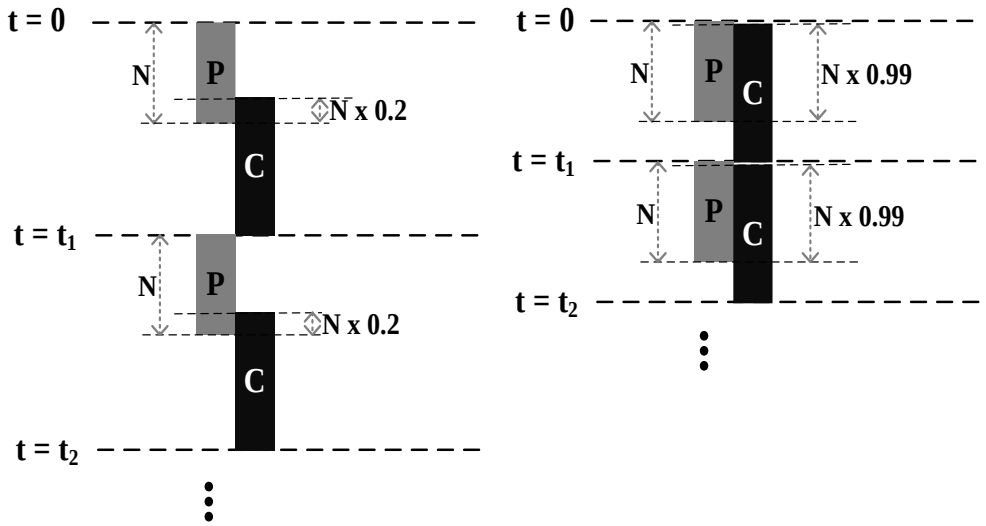

Figure 3: Overlapping Threads

Another terminology for producer-consumer Overlapping Threads is "pipeline parallelism." In [31], authors denote that "if it is possible to overlap the executions of the producer and consumer, the resulting parallelism is called pipeline parallelism."

IV.II. HOT-VERTICES TECHNIQUE

In a given sparse input, when the Overlapping Threads technique is applied, the highest degree vertices may cause a significant amount of memory requests. The consumer threads assigned to highest degree vertices will have higher data dependencies which in return will contribute significantly to the amount of memory requests done during the execution. This is because the fine-grain full-empty bit data structure is checked by each of the neighbors of a vertex and is may be checked repeatedly until the vertex is produced. The high-degree vertices are therefore much more likely to be accessed numerous times if they are not produced early, leading to significant extra bandwidth just for checking their

full/empty status. The Hot-Vertices technique produces the highest degree vertices early so that the polling done on these nodes is significantly reduced.

To lessen the overhead due to finding the high-degree vertices, I apply the *approximate sorting on the GPU* to determine the vertices with highest neighbors to lessen the time spent (=overhead) for analyzing the vertices. The overheads are counted for in my calculations. The input graph and its structure otherwise are not modified in any way.

As mentioned at the beginning of this section, the Hot-Vertices idea is explored with Overlapping Threads as an auxiliary method to improve the efficiency of the Overlapping Threads method. An illustration of Hot-Vertices is given in Figure 4 with 60% overlap. In Figure 4, on the left side only Overlapping Threads with 60% overlap is shown, and on the right, I apply the Hot-Vertices technique. The time t'_1 includes the time spent for sorting the vertices, and producing the top 20% of all nodes with highest degrees. The only overhead is the time spent on sorting and finding out In the overall diagram, I try to evaluate if $t'_4 < t_2$ by varying the amount of early-produced hot vertices among 2%, 10%, and 20%.

IV.III TASK SWITCHER TECHNIQUE

Every parallel algorithm starts with a pre-defined task assignment for the kernels. A kernel may sometimes be more suitable for GPU execution and other times more suitable for CPU execution. Generally, when parallelism is higher GPU execution is preferred, though there are other factors that influence the optimal decision of placing a kernel on the GPU or CPU. Conditions, such as the amount of available parallelism and the usage of resources, may change during execution. My work is not concerned about the latter since

it often requires a run-time solution [15] which checks the resource use on-the-fly which I do not aim to provide with this work. Rather, a compile-time tuning decision is made for the former, that is, the amount of available parallelism.

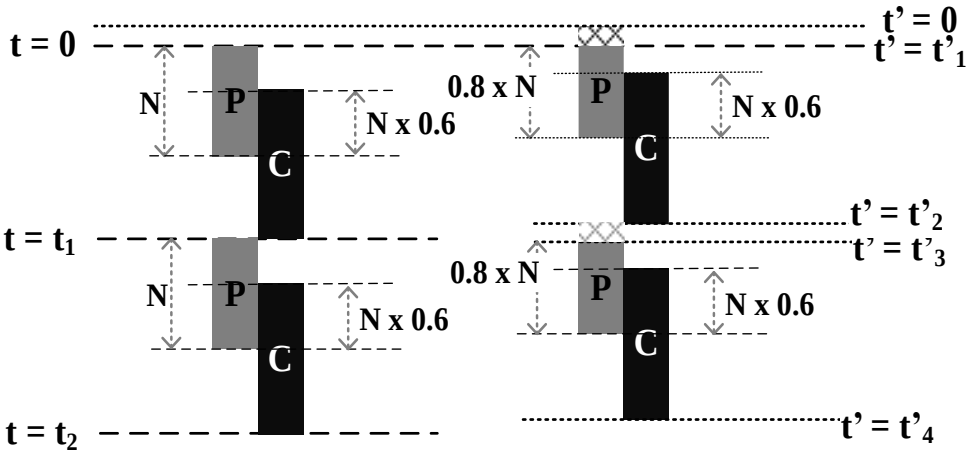

Figure 4: Overlapping Threads combined with Hot-Vertices

The Task Switcher is a framework for making dynamic decisions on whether a kernel should execute on the GPU or CPU component of the heterogeneous chip using a combination of compile-time and run-time information. The framework has two main components. The first is a programming pattern and helper scripts that enable an auto-tuned per-kernel decision. Each kernel is wrapped with a conditional to test where it should be run. The condition depends either on a static decision communicated via a script or a dynamic decision that depends on available parallelism during execution. The second framework component statically determines the parallelism breaking point: when expected available parallelism is above the breaking point, the kernel executes on the GPU and executes on the CPU otherwise. The breaking point is computed based on a model that takes into account the relative frequencies of the CPU and GPU, the performance ratio

between the two core types when executing vector instructions, the fraction of vector instructions in the kernel, and possible additional programmer-provided hints. The model is described in detail in Section IV.III.I.

To make a dynamic decision, the programmer must estimate available parallelism and compare to the breaking-point value computed by the Task Switcher model. As a concrete example consider the MIS application where the length of the work list at the beginning of each iteration is a good indicator for parallelism and can be readily used to make a decision on where each kernel should execute every iteration. This process is depicted in Figure 5 with possible decisions on placement for two producer-consumer kernels denoted (P and C, respectively) shown in in Figure 6. Scenario A in Figure 6 shows an example where the producing kernel (P) is always assigned to the CPU and the consuming kernel (C) to the GPU. Scenario B illustrates the breaking point being reached after the first execution of kernel C, after which all kernels are assigned to the CPU. Scenario C illustrates another possibility where both kernels switch their preferred execution core type during execution.

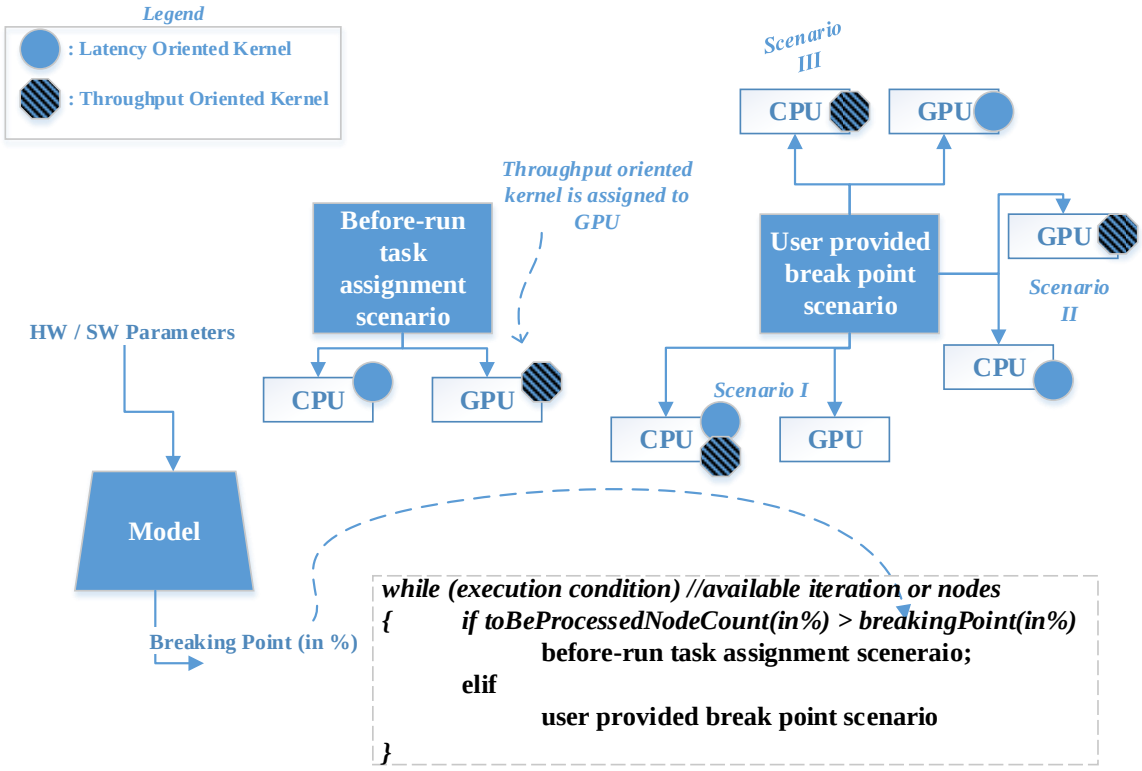

Figure 5: Illustration of Task-Switcher technique

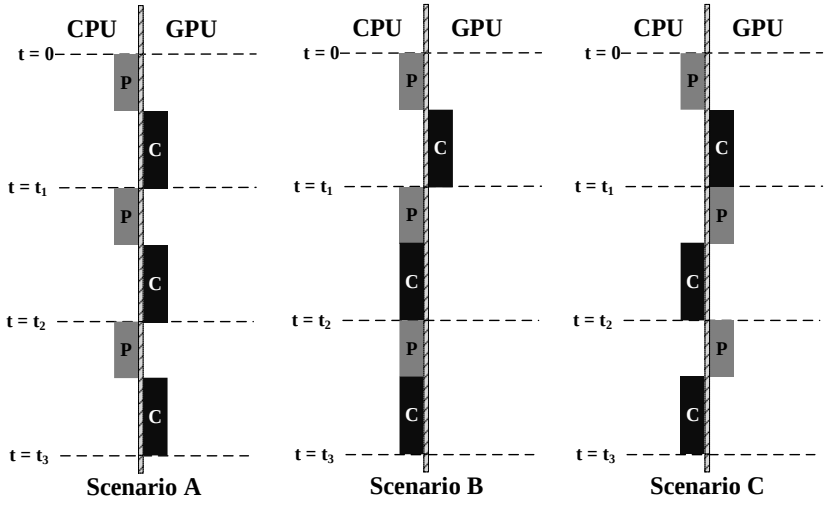

Figure 6: Task Switcher, example scenarios

IV.III.I Model of Task Switcher Technique

The model and decision maker (MDM) evaluation outcome is breakpoint percentage which is used to apply new task placements during the execution. It takes the following variables into its decision process:

- *Scalar / Vector Performance Factor:* Throughput-intensive and compute-intensive micro-benchmarks are run on both of the CPU and GPU cores separately, and a ratio is taken. This is independent of the algorithm.
- *Frequency Ratio:* The frequency ratio of CPU and GPU cores.
- *Kernel Hints:* Some of the kernels are known to be efficient on certain type of core and this information is given as a hint to the model by the user.
- *Kernel Details:* For each kernel, scalar and vector instructions are counted, and an approximate ratio is given to the model for each kernel.

This model has only one main duty: to provide a breakpoint percentage. The percentage is calculated with the following equation:

Equation 1: The percentage calculation, used for deciding on the time of applying the decisions

$$\% = 100 * \frac{1}{Freq\ Ratio(CPU, GPU) * (Scalar - Vector\ Perf\ Factor)} * \frac{1}{\sum Scalar / Vector\ Instr. Ratio}$$

Equation 1 provides a percentage so that a user-provided task placement decision can be applied. For every kernel, the ratio of scalar instruction count/vector instruction count is taken. This tuning parameter, which determines when to execute a kernel on the CPU and when on the GPU, is heuristically determined using a compiler analysis of the kernel code. For example for MIS, the model equation yields 26.3%, as can be seen in Equation 2, that is, "when the amount of available parallelism is decreased to 26.3%, the user defined task placement decision should take place." For MIS, this decision is "assign all the tasks to CPU", since user provides the fact that after a while, the available parallelism will be reduced and CPU would be much faster executing the remaining tasks. The numbers 9.12, 1.02, 0.89 denote the ratios of scalar to vector instructions in each kernel of MIS, calculated via a compiler analysis.

Equation 2: Evaluation of the equation for MIS

$$\% = 100 * \frac{1}{5.5 * \left(\frac{1}{16}\right)} \left[\frac{1}{9.12 + 1.02 + 0.89}\right] = 26.3\ \%$$

The user provides the information regarding what type of actions should be taken after the decision point has been reached. The information from the user is provided in the forms of knobs to a script which arranges and modifies the OpenCL code to be executed.

V. Methodology

To evaluate the ideas presented, I have utilized an actual heterogeneous chip, nine different sparse inputs from 10th DIMACS Implementation Challenge [32] and two producer-consumer algorithms.

V.I. Algorithms and Inputs

Two different algorithms are picked due to their importance among the graph algorithms and their suitable behavior for my evaluations:

1. Maximal Independent Set
2. Preflow-Push

V.I.I Maximal Independent Set (MIS)

Maximal independent set (MIS) [21] in an undirected graph is a maximal collection of vertices, I, which are subject to the restriction that no pair of vertices in I are adjacent. MIS is commonly used as a basic block in many application domains such as pattern recognition, computer vision and molecular biology. The serial implementation of the algorithm is given in Figure 7.

```
S = empty set,  C = V,
while C is not empty
{
  label each v in C with a random r(v),
  for all v in C in parallel
        {
              if r(v) < min( r(neighbors of v) )
              {
              move v from C to S,
              remove neighbors of v from C,
              }
        }
}
```

Figure 7: Serial maximal independent set

The parallel implementation of MIS starts with labeling each vertex with a random value and each vertex during an iteration, where a thread is assigned to, decides if it can be included in the set or not. Depending on the outcome of the decision the vertex will be added to the current set. For the vertices added to the set of the current iteration, the algorithm marks all neighbors of the selected vertices inactive. This removes the neighboring vertices from the candidate list and disallows them from participating at the next iteration of the algorithm. The algorithm terminates when all nodes are visited and evaluated.

Parallel implementation of MIS is given in Figure 8 and its kernels are given in Figure 9. This code shows how the Task Switcher enables kernel execution on a particular core type. Please note that in Fig 8., ts_<kernel>_perct is the variable name for the percentage calculated in Task Switcher. If the Task Switcher is not enabled the SVM functionality allows us to overlap the kernels without any barrier between kernels. Please also note that when we would like to enable Overlapping Threads the definitions for the

variables used during the execution should be different. This is again provided with basic if-else blocks in my code to be able to switch to Overlapping Threads technique.

```
S = empty set,  C = V,
while C is not empty
{
        if(taskSwitcherEnb && ts_randomize_perct){
                Call Randomize() kernel on GPU
        } else {
                Call Randomize() kernel on CPU
        }
        if(taskSwitcherEnb && ts_mis_perct){
                Call Randomize() kernel on CPU
        } else {
                Call Randomize() kernel on GPU
        }
        if(taskSwitcherEnb && ts_randomize_perct){
                Call Deactivate() kernel on CPU
        } else {
                Call Deactivate() kernel on GPU
        }
}
```

Figure 8: Parallel MIS implementation

```
kernel MIS(){
        execute[I] = true
        if(node I is active) {


                for every neighbor of I {
                        while(!ready[neighbor]){
                        }
                        if(randomize[I] > randomize[neighbor] ){
                                execute[I] = false
                        }
                }
        } else {
                execute[I] = false
        }
}


kernel Randomize(){
        randomize[i] = copyFromRandStream()
        ready[i] = True
}

kernel Deactivate(){
        if(execute[I] is true){
                I is selected
                Remaining nodes--
                for every neighbor of I{
                        if (neighbor is inactive){
                                Remaining nodes--
                        } elif (neighbor is active){
                                Neighbor is inactive
                                Remaining nodes -
                        }
                }
        }
}
```

Figure 9: The kernels of MIS

V.I.II Preflow - Push (PP)

Given a source node and a sink node in an undirected graph and given capacity constraints on the edges, the maximum flow problem tries to maximize the amount of flow that can be sent from source to sink. One of the solutions to maximum flow problem is the Preflow-push algorithm.

In the algorithm, nodes are temporarily allowed to have excess flow, i.e., they might have more incoming than outgoing flow. Such nodes are called *active nodes.* Another characteristic of the algorithm is that every node is assigned a height, so that nodes can only send flow to nodes with a lower height than their own. The algorithm is initialized by assigning a height of N (the number of nodes) to the source and zero to the sink and all of other nodes.

```
Workset ws = new Workset(g.getSource()),
        foreach (Node node: ws) {
                g.relabel(node)
                for (Node neighbor : graph.getNeighbors(node)) {
                        if (graph.pushFlow(node, neighbor) > 0) {
                                if (!neighbor.isSourceOrSink())
                                        ws.add(neighbor),
                                if (node.excess() <= 0)
                                break,
                        }
                }
                if (node.excess() > 0)
                ws.add(node),
        }
```

Figure 10: The serial version of PP

Preflow-Push is used to solve problems such as disjoint paths and bipartite matching, as one of the main blocks to such graph algorithms. The Preflow concept was introduced by Karzanov ($O(V^3)$), and later Goldberg et al. designed an algorithm with

$O(V^2E)$. Among the many approaches, Goldberg's implementation is relatively easier to parallelize than other augmenting path-based algorithms, which is my base.

The parallel version of PP starts with initializing the height, edge capacities, and excess arrays. Its pseudo-code is given in Figure 11. Height is used for find the direction of flow and excess denotes which node has excess flow. Edge capacity as the name implies defines the capacity of each edge in a given graph. First kernel, Preflow(), finds the neighbor with the lowest height and pushes the excess of the main node (at where a thread is assigned) or capacity of the edge between a main node and selected neighbor, whichever is the minimum. To eliminate race conditions, I have another kernel called push-adjust(), which adjusts the excess and capacities. The BFS stage is for adjusting the heights of the nodes starting from the sink. The details of the BFS are not discussed and I use the two-stage parallel BFS implementation given in [36].

```
initialize height, edge capacity, excess arrays
while(sink does not have excess){
        kernel Preflow() on GPU
        kernel Push-Adjust() on CPU
        if (first iteration){
                kernel BFS() on GPU
        } else {
                kernel BFS() on CPU
        }
}
kernel Preflow(){
        if (node I has excess) {
                find the neighbor with lowest height
                if (selected neighbor height < height of I) {
                        find minimum(excess[I],edge capacity)
                        excess[selected neighbor] += min
                        capacity[edge] -= min
                        excess[I] -= min
        }
}
kernel Push-Adjust() {
        for every neighbor of I{
                if(height[I] > height[neighbor]){
                        if(excess[I] >= capacity[ToNeighbor]){
                                excess[I] -= capacity[ToNeighbor]
                                excess[neighbor] += capacity[ToNeighbor]
                                capacity[ToNeighbor] = 0
                        } elif (excess[I] < capacity[ToNeighbor]){
                                excess[I] = 0
                                excess[neighbor] - = excess[I]
                                capacity[ToNeighbor] -= excess[I]
                        }
                }
        }
}
// BFS implementation is taken from [36].
```

Figure 11: The parallel version of PP and its kernels

V.I.III Inputs

Nine different sparse graph inputs are picked from 10th DIMACS Implementation Challenge.

Table 1: Inputs used for evaluation.

Input Name	*# of Vertices*	*# of Edges*
cage15	5,154,859	94,044,692
nlpkkt120	3,542,400	46,651,696
kkt_power	2,063,494	6,482,320
g3_circuit	1,585,478	3,037,674
thermal2	1,227,087	3,676,134
ecology1	1,000,000	1,998,000
ldoor	952,203	22,785,136
audikw_1	943,695	38,354,076
af_shell9	504,855	8,542,010

V.II HARDWARE

For the evaluations, I have utilized *AMD APU A10-7850K APU*. The APU has 4 CPU cores (28nm Steamroller core) at *4 GHz* frequency and 8 GCN (Graphics core next) cores which provide 512 GPU threads in total at *720 MHz* clock frequency. The system can share the virtual memory between CPU and GPU, allowing them to have uniform access to the entire memory region.

My system has *8GB DDR3-2133* memory in two channels and peak bandwidth from OpenCL micro-benchmark “Global Bandwidth” with vector width 8 is *12.75 GB/s*

for CPU random read and *51.10 GB/s* for GPU random read. Although the bandwidth calculation methodology is not detailed in the paper, [33] provides 7.8 GB/s for CPU and 28.9 GB/s GPU as peak bandwidth with the same APU with 32GB DDR3 memory. I suspect that the memory was not in dual channel mode and that's why almost 2x difference exists.

VI. Evaluation

VI.I Maximal Independent Set

VI.I.I Overlapping Threads

Overlapping Threads could be applied to MIS since MIS provides kernels with data dependencies between them. When Overlapping Threads technique is applied to MIS with the inputs I have, it achieves a speedup of 1.3x to 1.6x as seen in Figure 13. During the evaluations, I have observed that after ~80% of the overlap there is no execution time gain and I claim this is the sweet-spot for this algorithm.

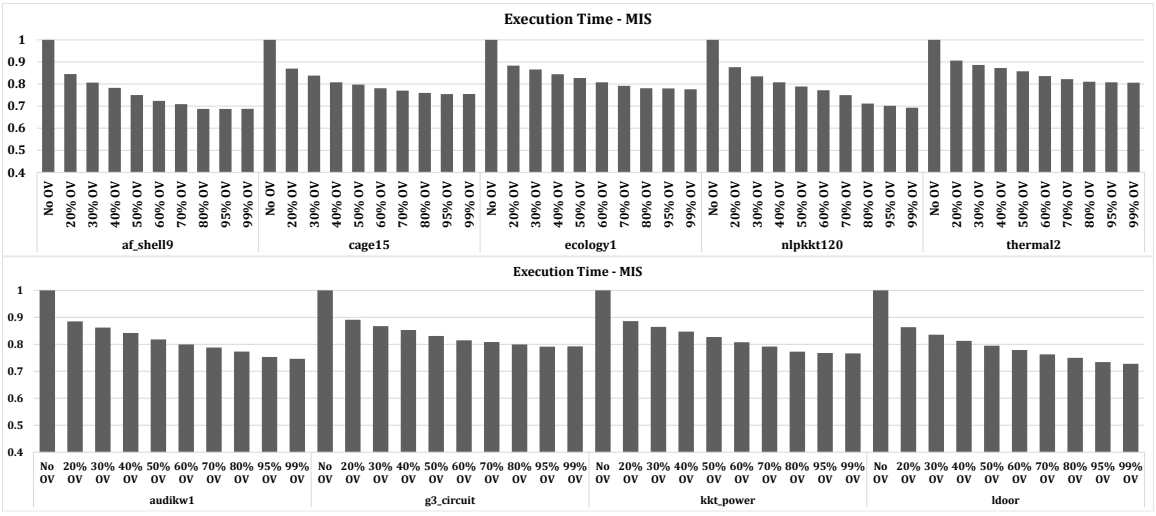

Figure 12: Overlapping Threads applied to MIS.

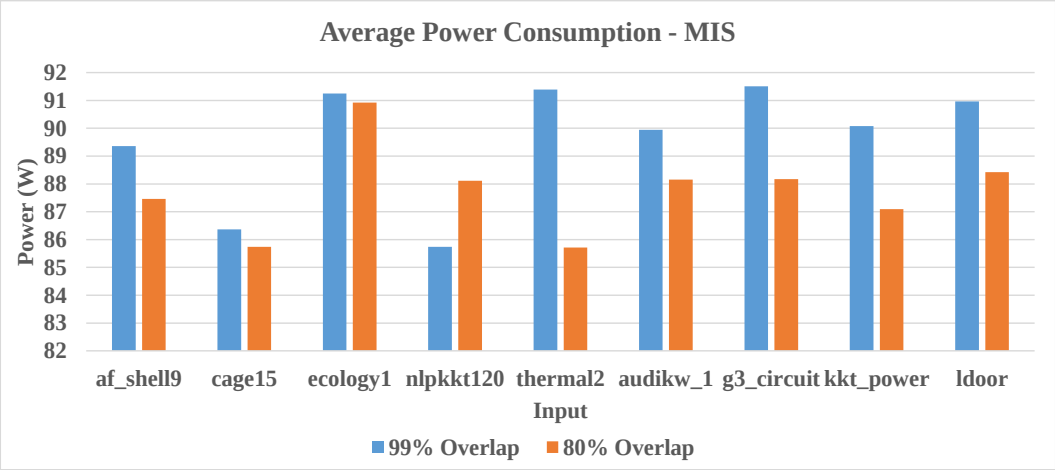

Figure 13: Average power consumption for MIS for 99% and 80% overlap.

Please note that it is relatively easy to define a percentage for the framework and the framework would provide the code when the Overlapping Thread is enabled. One would prefer applying 99% overlap, and it would give the same speed-up as 80% overlap. However, this would result in higher energy consumption since there would be more polling done in 99% overlap case. To back-up my claim I have measured the power consumption of the chip for the 80% and 99% overlapping amounts and found out that 80% overlapping consumes less power in 8 out of 9 inputs as shown in Figure 13.

VI.I.II Overlapping Threads and Hot-Vertices

Although any node can contribute to the amount of polling done throughout the execution, the most significant impact is done by the nodes with highest degrees. Hot-Vertices technique aims to reduce the amount of polling done by the threads assigned to the nodes with highest degrees by pre-producing such nodes so that the polling done on these nodes would not be needed. During my evaluations with MIS, I prove that producing such vertices early reduces the overall execution time. For simplicity and readability, some

of the results are not shown in Figure 14, and only essential data is presented for comparison purposes: No overlap and 99% overlap with different Hot-Vertices amount. For each of the inputs, I calculate no Hot-Vertices (HV), 2%, 10% and 20% HV. The percentages denote how many of the highest degree vertices are produced early. As seen in Figure 14, the Hot-Vertices idea reduces the total execution time, which also incorporates overheads such as the time spent for finding the Hot-Vertices.

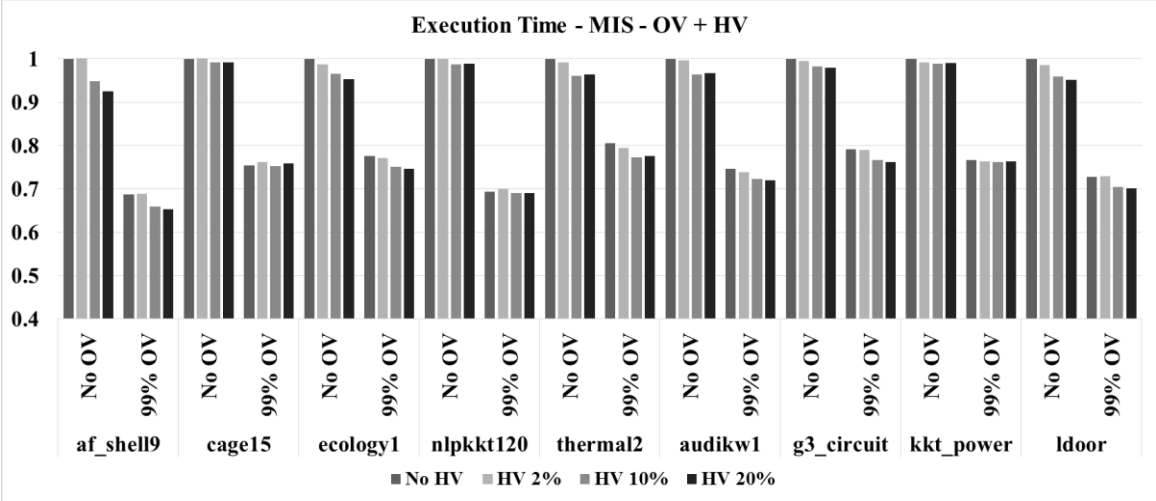

Figure 14: MIS with Overlapping Threads and Hot-Vertices

VI.I.III Task Switcher

When the Task Switcher is enabled in the framework, model evaluation takes place at compile time and task placement decisions are applied dynamically to MIS. For MIS, the Task Switcher achieves ~1.6x to 1.9x speed-up compared to CPU-only execution as shown in Fig 15.

Implementation-wise, MIS has three kernels and each kernel is assigned to either CPU or GPU. The task placement done by the model and decision maker is shown in Fig 16.

Each colored block denotes that that particular kernel will be executed on CPU or GPU depending on the location of the block. For example for af_shell9, at the first computational step Randomize kernel executes on CPU while MIS and Deactivate kernel executes on the GPU. At the second computational step, Randomize and MIS execute on CPU, while the Deactivate kernel will be executed on GPU. These decisions are given by the user, and the Task Switcher decides *when* to apply these decisions by calculating the equation as discussed in Section IV.III.I. Please note that after 26.3% reduction happens for every input, the tasks on GPU are assigned to CPU.

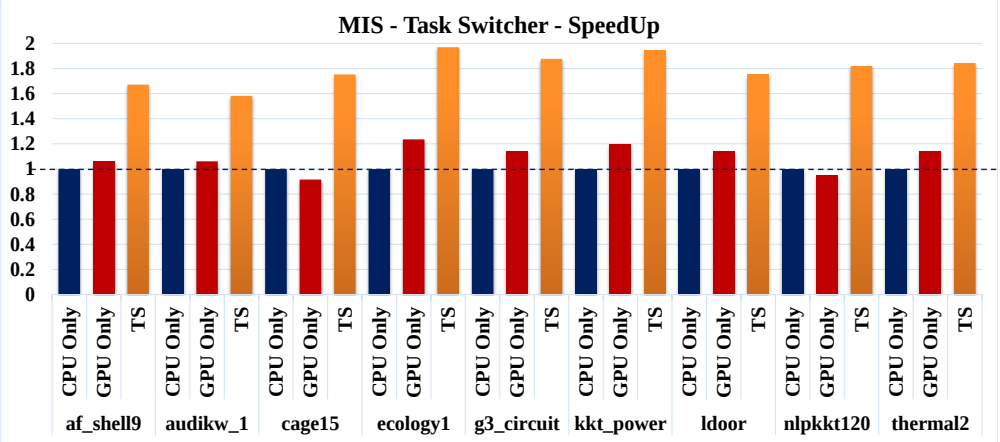

Figure 15: Task Switcher speed-up for MIS

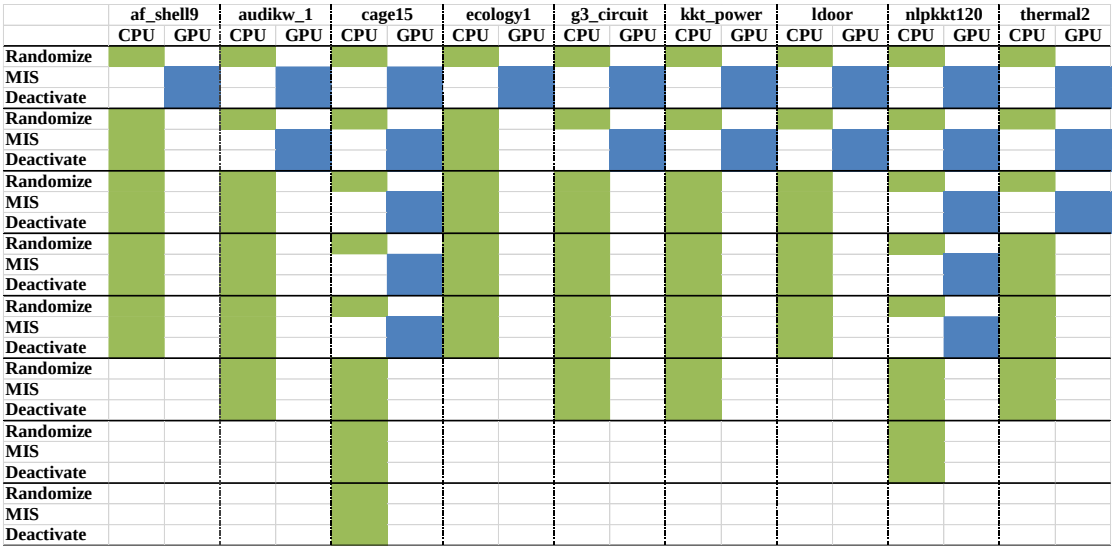

Figure 16: The task placement decisions made by the Task Switcher model and decision maker for MIS

VI.II PREFLOW - PUSH

Preflow-push shows different behavior than MIS. Firstly, it does not have the strong producer-consumer interaction that MIS does. It has three main steps namely: Preflow, Push Adjust, and BFS (see Figure 11). These steps do not exhibit strong data dependency among them, which makes the Overlapping Threads technique exploration for Preflow-push infeasible. Secondly, the amount of parallelism available in PP after the first computational step is significantly reduced. This difference between the first iteration and the rest, makes Preflow push benefit significantly from task switching. Even though there is no readily-available variable for estimating available parallelism as with MIS, the programmer can use the Task Switcher framework and helper scripts to provide hints and directives on where each kernel should execute. My analysis easily indicates that the Preflow kernel should always execute on the GPU, Push Adjust on the CPU, and that BFS

should run on the GPU in the first iteration but on the CPU in following iterations where parallelism is lower. These decisions are expressed in a tuning file that is then used by a script and the OpenCL framework code within the application.

VI.II.I Task Switcher

Preflow-push has three kernels, and for each kernel, certain decisions are applied by the Task Switcher model & decision maker:

- BFS at first computational stage is assigned to the GPU cores and the remaining BFS at the next computational stages are assigned to the CPU.
- PP is assigned to GPU
- Adjust is assigned to CPU

I run the given code by the framework on my heterogeneous chip and compare the results to the CPU-only execution of the Preflow-Push. My technique on Preflow-Push provides 1.3x as minimum and 18x as maximum speed-up. Excluding two inputs, as can be seen in Figure 18 the speed-up varies between 1.3x to 2.4x, averaging 1.78x speed-up. The reason I exclude two of the inputs is that they take very few computational steps and BFS calculation takes a significant portion of the total execution time of Preflow-Push, resulting in significant speed-up as shown in Figure 18 when BFS portion is accelerated via GPU. This is solely dependent on the structure of these sparse graphs. In Figure 18, the CPU-only execution is normalized to one, and the theoretical best and Task Switcher results are compared.

	af_shell9			audkw_1			cage15			ecology			g3_circuit			kkt_power			ldoor			nlpkkt120			thermal2		
	Total	CPU	GPU	Total	CPU	GPU	Total	CPU	GPU	Total	CPU	GPU	Total	CPU	GPU	Total	CPU	GPU	Total	CPU	GPU	Total	CPU	GPU	Total	CPU	GPU
Preflow (GPU)	84	0	84	24	0	24	5	0	5	11	0	11	4	0	4	8	0	8	37	0	37	39	0	39	83	0	83
Push-Adjust (CPU	84	84	0	24	24	0	5	5	0	11	0	11	4	0	4	8	1	7	37	37	0	39	39	0	83	0	83
BFS	84	82	2	24	11	13	5	0	5	11	10	1	4	3	1	8	7	1	37	34	3	39	9	30	83	80	3

Figure 17: The required assignments for the best theoretical speed-up.

I also measure the best possible theoretical speed-up by running the code on CPU and GPU and finding the minimum execution time for each of the kernels at each computational step and the information is provided in Figure 17. In Figure 17, the gray boxes denote where my Task Switcher technique, when applied to PP, has failed to achieve the best task assignment. For example for BFS & af_shell9, the best theoretical task assignment was to assign first two iterations to GPU, but my framework assigns only the first iteration to GPU as rest is executed on CPU. I prove that my framework can achieve 84% accuracy while making task placement decisions compared to best theoretical speed-up.

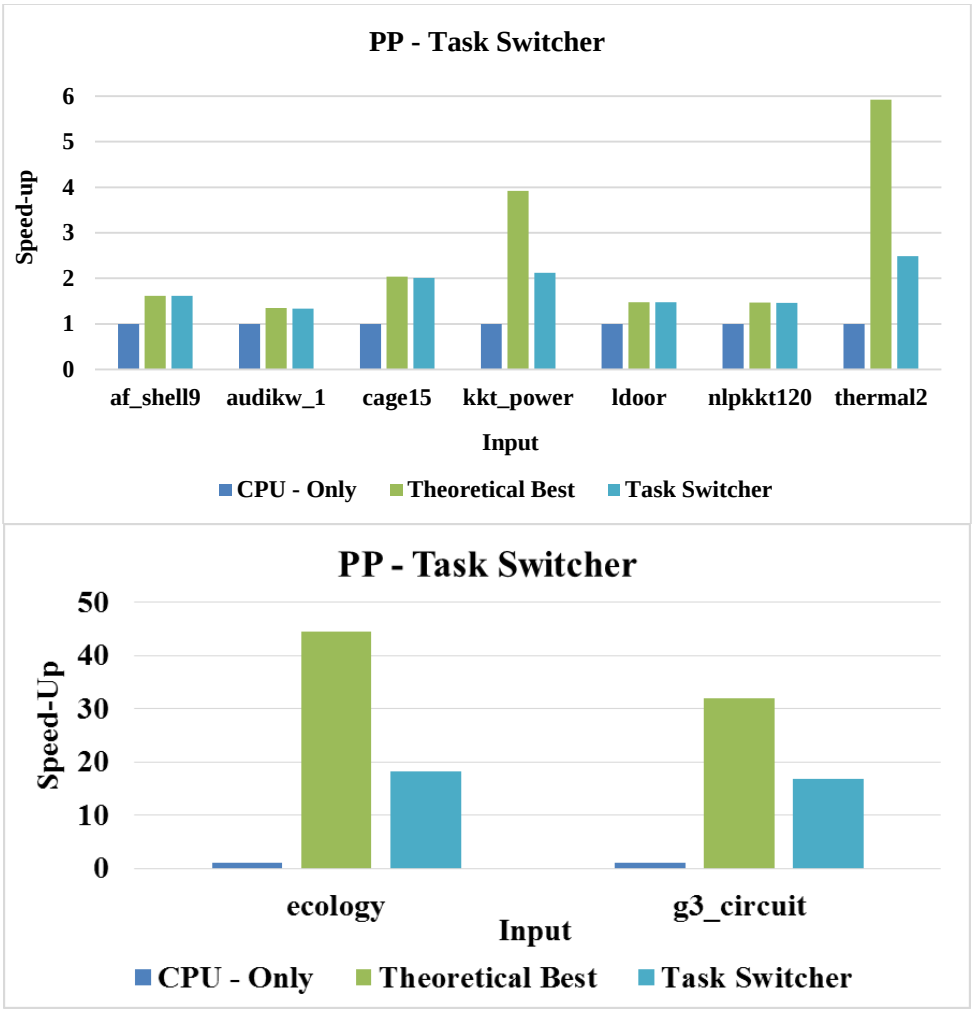

Figure 18: Preflow-push in Task Switcher

VII. Conclusion

The use of heterogeneous chip multiprocessors for improving the performance and energy efficiency of computations is an active research topic. I provide insights into how to utilize such processors for two commonly used sparse graph algorithms and demonstrate how to accelerate these applications on a commodity heterogeneous Accelerated Processing Unit that combines latency-oriented CPU cores and throughput-oriented GPU cores. I evaluate how the combination of overlapping the execution of data-dependent kernels (Overlapping Threads), reducing the amount of memory polling needed to accomplish this overhead (Overlapping Threads + Hot-Vertices), and applying a heuristic for task placement that is based on static analysis (the Task Switcher), provides substantial improvements in execution.

VIII. References

[1] T. Mudge, "Power: a first-class architectural design constraint," *Computer,* vol. 34, no. 4, pp. 52-58, 2001 April.

[2] J. A. Joao, A. Suleman, O. Mutlu and Y. N. Patt, "Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs," *ISCA '13 Proceedings of the 40th Annual International Symposium on Computer Architecture,* pp. 154-165, 2013.

[3] A. Suleman, O. Mutlu, M. Qureshi and Y. N. Patt, "An Asymmetric Multi-Core Architecture for Accelerating Critical Sections," The University of Texas at Austin, Austin, TX, September 2008.

[4] H. Xie, "Get Ready For Next Generation APU Architecture," Advanced Micro Devices, 2013.

[5] A. M. Devices, "AMD “Carrizo” architecture unveiled at ISSCC," Advanced Micro Devices, [Online]. Available: http://www.amd.com/en-us/who-we-are/corporate-information/events/isscc. [Accessed 1 October 2016].

[6] I. Corporation, "Products (Formerly Sandy Bridge)," Intel Corporation, [Online]. Available: http://ark.intel.com/products/codename/29900/Sandy-Bridge#@All. [Accessed 25 10 2016].

[7] J. Lee and H. Kim, "TAP: A TLP-Aware Cache Management Policy for a CPU-GPU Heterogeneous Arhitecture," *HPCA '12 Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture,* pp. 1-12, 2012.

[8] I. Bratt, "HSA Queueing," Hot Chips, August 2013.

[9] M. Schulte, M. Ignatowski, G. H. Loh, B. M. Beckmann, W. C. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S. K. Reinhardt and G. Rodgers, "Achieveing Exascale Capabilities Through Heterogeneous Computing," *IEEE Computer Society,* pp. 26-36, 2015.

[10] M. Kulkarni, M. Burtscher, C. Cascaval and K. Pingali, "Lonestar: A Suite of Parallel Irregular Programs," *Performance Analysis of Systems and Software, 2009. ISPASS 2009,* 2oo9.

[11] S. Che, B. M. Beckmann, S. Reinhardt and K. Skadron, "Pannotia: Understanding Irregular GPGPU Graph Applications," *Workload Characterization IISWC 2013 IEEE International Symposiom on,* 2013.

[12] J. Feo, O. Villa, A. Tumeo and S. Secchi, "Irregular applications: architectures and algorithms," *In Proceedings of the First Workshop on Irregular Applications: Architectures and Algorithms,* pp. 1-2, 2011.

[13] J. Feo, D. Harper, S. Kahan and P. Konecny, "ELDORADO," *CF'05 Proceedings of the 2nd Conference on Computing Frontiers,* pp. 28-34, 2005.

[14] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie and D. Yeung, "The MIT Alewife Machine: Architecture and Performance," *ISCA '95 Proceedings of the 22nd annual international symposium on Computer architecture,* pp. 2-13, 1995.

[15] S. Panneerselvam and M. Swift, "Rinnegan : Efficient Resource Use in Heterogeneous Architectures," *PACT'16,* 2016.

[16] H. Wang, R. Singh, M. Schulte and N. S. Kim, "Memory Scheduling Towards High Throughput Cooperative Heterogeneous Computing," *PACT'14,* 2014.

[17] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. Hill, S. Reinhardt and D. A. Wood, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," *MICRO-46,* 2013.

[18] "OpenCL : The open standard for parallel programming of heterogeneous systems," Khronos, [Online]. Available: https://www.khronos.org/opencl/. [Accessed 1 October 2016].

[19] P. Raghavendra, "OpenCL 2.0: Fine-Grain Shared Virtual Memory," Advanced Micro Devices, 15 January 2015. [Online]. Available: http://developer.amd.com/community/blog/2015/01/15/opencl-2-0-fine-grain-shared-virtual-memory/. [Accessed 1 October 2016].

[20] J. Cohen and P. Castonguay, "Efficient Graph Matching and Coloring on the GPU," NVIDIA - GPU Tech Conference, 2012.

[21] M. Luby, "A Simple Parallel Algorithm for the Maximal Independent Set Problem," *Society for Industrial and Applied Mathematics,* vol. 15, no. 4, pp. 1036-1053, 1986.

[22] W.-m. Hwu, GPU Computing Gems Jade Edition, Elsevier, 2011.

[23] Arvind, R. S. Nikhil and K. K. Pingali, "I-structures: data structures for parallel computing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 11, no. 4, pp. 598-632 , 1989 .

[24] M. Noakes, D. Wallach and W. J. Dally, "The J-Machine Multicomputer : An Architectural Evaluation," *Computer Architecture, 1993., Proceedings of the 20th Annual International Symposium on*, 1993.

[25] J. Kim and C. Batten, "Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklist," *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, 2014.

[26] J. Ributzka, Y. Hayashi, J. B. Manzano and G. R. Gao, "The elephant and the mice: the role of non-strict fine-grain synchronization for modern many-core architectures," *ICS '11 Proceedings of the international conference on Supercomputing*, pp. 338-347 , 2011.

[27] J. P. Grossman, J. S. Kuskin, J. A. Bank, M. Theobald, R. O. Dror, D. J. Ierardi, R. H. Larson, U. B. Schafer, B. Towles, C. Young and D. E. Shaw, "Hardware support for fine-grained event-driven computation inton 2," *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems* , pp. 549-560 , 2013.

[28] A. Gontmakher, A. Mendelson and A. Schuster, "Using fine grain multithreading for energy efficient computing," *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 259-269 , 2007 .

[29] W. Zhu, V. Sreedhar, Z. Hu and G. R. Gao, "Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures," *Proceedings of the 34th annual international symposium on Computer architecture* , pp. 35-45 , 2007.

[30] Khronos, "OpenCL 2.0 Reference Pages," Khronos, [Online]. Available: https://www.khronos.org/registry/cl/sdk/2.0/docs/man/xhtml/. [Accessed 22 December 2016].

[31] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Mendez-Lojo, D. Prountzos and X. Sui, "The Tao of Parallelism in Algorithms," *PLDI'11*, pp. 12-25, 2011.

[32] D. A. Bader, H. Meyerhenke, P. Sanders and D. Wagner, "Graph Partitioning and Graph Clustering. 10th DIMACS Implementation Challenge Workshop.," in

American Mathematical Society and Center for Discrete Mathematics and Theoretical Computer Science, Georgia Institute of Technology, Atlanta, GA., 2012, February.

[33] J. He, S. Zhang and B. He, "In-Cache Query Co-Processing on Coupled CPU-GPU Architectures," *Proceedings of the VLDB Endowment,* vol. 8, no. 4, pp. 329-340, 12/2014.

[34] R. Ribeiro, J. Barbosa and L. P. Santos, "A Framework For Efficient Execution of Data Parallel Irregular Applications on Heterogeneous Systems," *Parallel Procesing Letters 25,* June, 2014.

[35] W. Song, S. Mukhopadhyay and S. Yalamanchili, "Amdahl's Law for Lifetime Reliability Scaling in Heterogeneous Multicore Processors," *2016 IEEE International Symposium on High Performance Computer Architecture,* 2016.

[36] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Hang-Ha Lee, Kevin Skadron "Rodinia: A benchmark suite for heterogeneous computing" *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on, 4-6,* October, 2009.

[37] AMD Graphics Core Next(GCN) Architecture, White Paper, June 2012

[38] Min Kyu Jeong, Mattan Erez, Chander Sudanthi, Nigel Paver "A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC," *Proceeding DAC '12 Proceedings of the 49th Annual Design Automation Conference*, Pages 850-855, June 2012

[39] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, Onur Mutlu, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," *Proceeding ISCA '12 Proceedings of the 39th Annual International Symposium on Computer Architecture*, Pages 416-427, June 2012