저작자표시-비영리-변경금지 2.0 대한민국

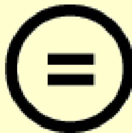
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

M.S. THESIS

Parallelism Management for Memory-Intensive Applications on Integrated CPU/GPU Architecture

CPU/GPU 통합 아키텍쳐에서의 메모리 집약적 어플리케이션에 대한 병렬성 관리

AUGUST 2021

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

박 지 연

Parallelism Management for Memory-Intensive Applications on Integrated CPU/GPU Architecture

CPU/GPU 통합 아키텍쳐에서의 메모리 집약적 어플리케이션에 대한 병렬성 관리

지도교수 Bernhard Egger

이 논문을 공학석사 학위논문으로 제출함

2021 년 04 월

서울대학교 대학원

컴퓨터공학부

박 지 연

박지연의 공학석사 학위논문을 인준함

2021 년 06 월

위 원 장	이 재 진
부위원장	Bernhard Egger
위 원	김 진 수

Abstract

Integrated architectures combine CPU and GPU cores onto a single processor die. Thanks to the shared memory architecture, costly memory copies to the GPU device memory can be avoided, and programmers can use both CPU and GPU cores to accelerate their applications. Especially for memory-intensive applications, however, utilizing all available core resources does not always maximize performance due to congestion in the shared memory system. Tuning an application to use the optimal number of CPU and GPU compute resources is a difficult and machine-dependent task.

This thesis presents an automated system that auto-tunes an OpenCL kernel for a given integrated architecture on the fly. A light-weight compiler extract the characteristics of a kernel as it is submitted to the OpenCL runtime. We then use a software-based technique to automatically rewrite the kernel to only utilize the compute resources that are expected to maximize performance based on a model. Our analysis shows that the memory access pattern and the cache utilization are the decisive factors when determining the parallelism of a kernel. To accommodate for the varying properties of different integrated architectures, we employ machine learning to create models for different architectures. The models are trained with microbenchmarks that generate a large number of varying memory patterns and evaluated with OpenCL benchmarks from different benchmark suites.

While the presented approach shows good results for older integrated architectures such as Intel Skylake and AMD Kaveri, it currently still does not achieve satisfactory results on the more recent architectures Intel Comet Lake

and AMD Picasso. However the analysis performed on memory pattern still has some insight on it.

Keywords: Parallelism Mangaement, CPU-GPU Integrated Architecture, Code Analysis

Student Number: 2019-22899

Contents

Abstract **i**

Contents **iii**

List of Figures **v**

List of Tables **vii**

Chapter 1 Introduction **1**

Chapter 2 Background **4**

2.1 Integrated CPU-GPU Architecture 4

2.2 OpenCL Programming 5

Chapter 3 Implementation **8**

3.1 Overview 8

3.2 Kernel Modification 8

3.3 GPU Throughput Analysis 10

3.3.1 Intra-Thread 11

3.3.2 Inter-Thread 13

3.4 Prediction Model 16

3.4.1 Feature Selection 16
3.4.2 Model Training 17

Chapter 4 Experiment **19**
4.1 Throughput Measurement 19
4.2 Target Device 20
4.3 Micro-benchmark Results 21
4.4 Real Benchmark Results 21
4.5 Result Analysis 22

Chapter 5 Conclusion **24**

Bibliography **24**

요약 **28**

List of Figures

Figure 1.1 Normalized work group execution time of matrix accumulation kernel for varying number of CPU cores while running GPU on both kernel on an Intel Comet Lake system. 2

Figure 2.1 SoC and ring interconnect architecture of Intel i7 processor 6700K[9] 4

Figure 2.2 2D kernel execution 6

Figure 2.3 The Execution Unit(EU) of Intel GPU.[9] 7

Figure 3.1 Original kernel code assigning 2-d matrix one to another 9

Figure 3.2 Modified kernel code for throttling 9

Figure 3.3 Example for throttling code 10

Figure 3.4 Intra kernel 1 code 12

Figure 3.5 Normalized time for kernels with different amount of stride 13

Figure 3.6 Normalized time for kernel 1, 2 14

Figure 3.7 Normalized time for kernel 3 15

Figure 4.1 Performance with work group ending 20

List of Tables

Table 3.1 Kernel variable description 11

Table 3.2 Kernels for intra-thread analysis 11

Table 4.1 Results by Model-1 21

Table 4.2 Results by Model-2 21

Table 4.3 Real Benchmarks Prediction Results - Model 1 (normalized time) 22

Table 4.4 Real Benchmarks Prediction Results - Model 2 22

Chapter 1

Introduction

Many recent desktop and mobile processors integrate CPU and GPU cores onto a single die. Examples of such integrated architectures are AMD's accelerated processing units (APU) [1] and Intel processors since Skylake [5]. Integrated architectures present an attractive alternative to dedicated GPUs not only because of the lower cost and energy consumption [11, 13], but also because they allow for a simpler parallel programming model. Since the CPU and GPU cores share the off-chip memory, applications can exploit the processing power of CPUs and GPUs without requiring costly data copying operations.

Achieving maximal performance on integrated architectures for data-parallel workloads, however, is a challenging problem. Integrated architectures enable the CPU and the GPU to execute different tasks that collaborate through shared memory [15, 7]. The characteristics of a parallel workload determine its affinity: workloads with frequent control flow changes and irregular memory accesses tend to achieve better peformance on the CPU, while kernels with little control divergence and regular memory access patterns typically achieve

superior performance on the GPU [12, 16, 3]. Whether a workload is CPU or GPU affine, adding more processing cores of the other type intuitively leads to higher performance [10, 17]. As we will demonstrate shortly, not only is this not true, but orchestrating the co-execution of a workload on both CPU and GPU cores is a non-trivial problem by itself [17, 3] since the optimal static partitioning of the workload to CPU and GPU is unknown and a dynamic partitioning scheme needs to implement a low-overhead global shared queue to distribute the workload.

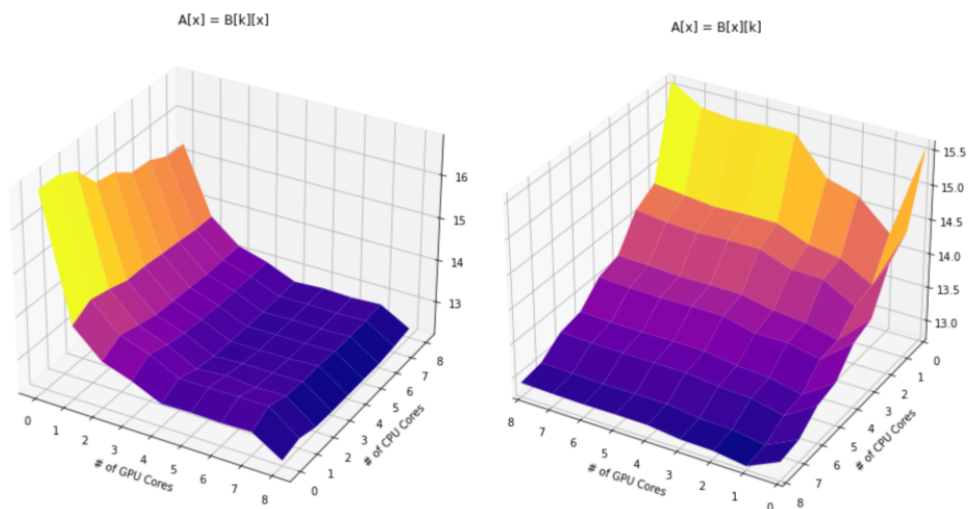


Figure 1.1: Normalized work group execution time of matrix accumulation kernel for varying number of CPU cores while running GPU on both kernel on an Intel Comet Lake system.

As stated earlier, utilizing all available computational resources does not necessarily lead to maximum performance. Figure 1.1 visualizes the single work group execution time of two different kernels, which reads from two different matrix and writes it to another. On left kernel, best case is using all for GPU

and less(2) for CPU, however on right kernel, the best case is to use less cores for GPU and all for CPU. On this simple example we can see that using all resources not always lead to best results.

Managing thread-level parallelism has long been an important technique for multi-core processors [14, 6, 2]. However, existing techniques are not applicable for integrated architectures. Unlike CPUs, GPUs do not allow for a flexible adjustment of the number of threads once a kernel is launched because the GPU threads are managed by the GPU's hardware scheduler. A number of techniques have been presented to select the optimal execution mode [17, 18, 4]. For example, a state-of-the-art technique [17] executes an application exclusively on the CPU, the GPU, or in parallel on all cores of the CPU and the GPU, based on a decision tree.

We tried to present a way to find and apply an appropriate degree of parallelism for integrated architectures. First, to manage the thread-level parallelism of the GPU, we transforms OpenCL kernels to be able to run with a specific number of threads. Also, we made a machine learning modeled based on results from micro kernels, and calculate performance-relevant static features of the OpenCL kernel. With pre-runned data with synthetic workload, we expect to predict optimal degree of parallelism on any architecture for target application before running it.

The remainder of this paper is organized as follows. In Section 2, we discuss the background information to extract features, about how GPU runs workload on it. In Section 3, we describe how we runs limited number of cores and the ML model to determine the number of threads for the CPU and the GPU. Sections 4 discuss the experimental setup and the results. Section 5 concludes this paper.

Chapter 2

Background

2.1 Integrated CPU-GPU Architecture

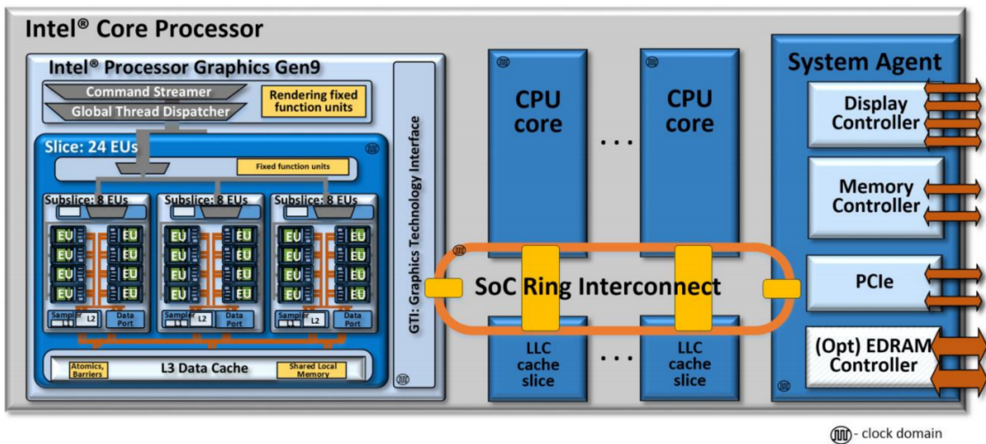


Figure 2.1: SoC and ring interconnect architecture of Intel i7 processor 6700K[9]

Unlike traditional GPU, which located outside of CPU chip and has separate memory, integrated systems shares same memory for both CPU and GPU.

Figure 2.1 shows Intel Gen9 architecture, which also shares last level cache with GPU. On discrete architecture, programmers should copy data from CPU memory to GPU before GPU start working, and copy it back when GPU completes it computation. For application in which CPU and GPU should communicate often the overhead of memory copying gets more significant. By using integrated systems we can avoid this overhead.

2.2 OpenCL Programming

In this section, we discuss the background of running kernels on GPU systems using OpenCL.

Based on OpenCL programming models, the workloads are consit of multiple work items. One work item is related to data position on which GPU thread will perform on. A *compute unit* in GPU will process multiple work items simultaneously on same instruction. Since GPU contains of several CUs, we make a group of work items to run on single CU. The number of work items can be larger than the number of cores inside CU since GPU can switch working data and use others in work group for performance. On OpenCL, we can number the work items in different dimension. The work item number and dimension will make memory accesses form differently.

On 1D kernel, the work items will be assigned continuously. Figure 3.1 is an example for 1D kernel. From the kernel, we can see that continuous threads in a work group will write on continuous region on array B, while starting read from N-strided on array A.

On 2D kernel, the work groups are shaped on 2D space. Figure 2.2 shows when the work group is size of 42. On example kernel, the global id of work items in a box will be numbered from 0 to 3 for *global_id(0)* and 0 to 1 for

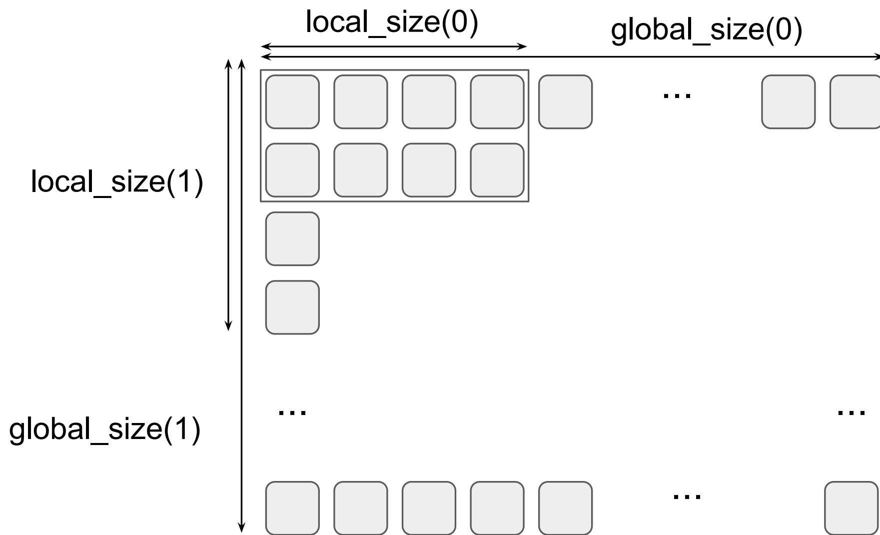


Figure 2.2: 2D kernel execution

global_id(1). We used these two values for indexing data region which the work item accesses, like *A[i][j]*, the size of work group affects on memory pattern of kernel.

Unlike AMD, Intel uses concept of threads which performs SIMD operation. Figure 2.3 illustrates intel Execution Unit. Each gen9 EU has seven threads and each thread has its own registers to deal with SIMD operations. The number of SIMD lanes is decided on compile time by intel graphics compiler.

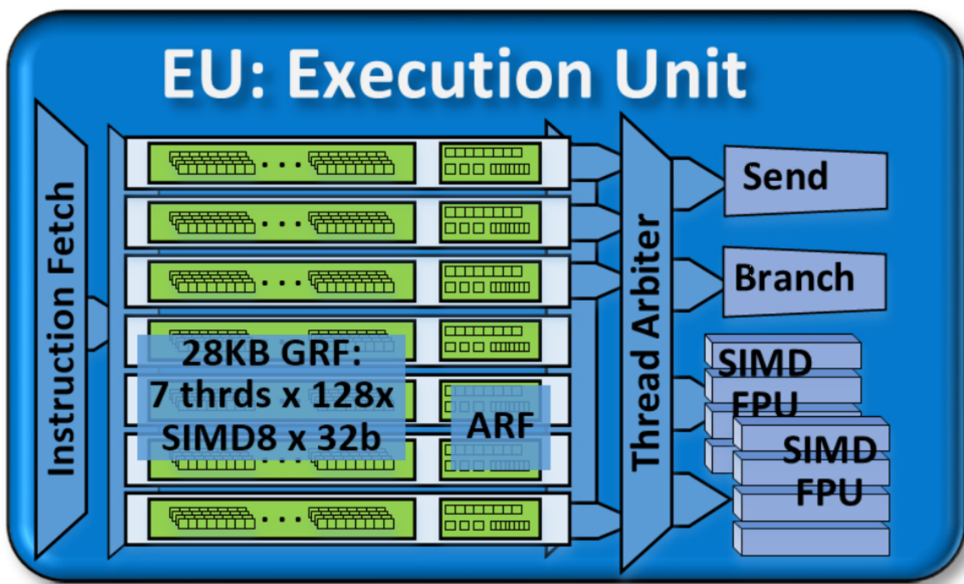


Figure 2.3: The Execution Unit(EU) of Intel GPU.[9]

Chapter 3

Implementation

3.1 Overview

3.2 Kernel Modification

This section discuss about how to throttle workloads. We only modified software code to leverage the number of running cores on devices. The number of activate core is dependent on two parameters: *throttling_alloc* and *throttling_mod*. Code for throttling is inserted before kernel loop starts. The if-else statement tests whether the *work item id % throttling_mod < throttling_alloc*. If not, core will skip the whole loop, not making any memory requests.

Figure 3.1 shows the original kernel code of simple applications which read a value from 2d matrix A and assign it to B. The index of matrix is calculated with *global_id* and loop variable j. We modify the original code to Figure 3.2. In throttling kernel, if-else statement with *throttling_alloc* decides whether to execute loop or not. Also we change initialization of i. To make other core to

```
__kernel void micro_kernel(__global float *A,
                           __global float *B,
                           int N)
{
    int x = get_global_id(0);

    for(int k = 0; k < N; k++)
        B[x] = A[x*N + k];
}
```

Figure 3.1: Original kernel code assigning 2-d matrix one to another

```
__kernel void micro_kernel_repeat(
                    __global float *A,
                    __global float *B,
                    int N,
                    int throttling_mod,
                    int throttling_alloc)
{
  int local_wi = get_local_id(0);

  if (get_local_id(0) % throttling_mod < throttling_alloc) {
     for(int work = local_wi;
             work < (local_wi + throttling_mod - (local_wi % throttling_mod));
             work += throttling_alloc) {
        int global_id_0 = get_group_id(0) * get_local_size(0) + get_global_offset(0) + work;
        int x = global_id_0;
        for(int k = 0; k < N; k++)
            B[x] = A[x*N + k];
      }
   }
}
```

Figure 3.2: Modified kernel code for throttling

run skipped work item, we make an local work item list and get work item id from the local list. If the core workload is skipped, it will not increase local work item list, so other core which not skipped will be assigned with next id and do work. Figure 3.2 illustrates the idea of throttling code. We can see the only half of total cores are running, if we set *throttling_alloc* half of *throttling_mod*. To avoiding skipping work items, we calculate next local work id which should be processed by stopped core.

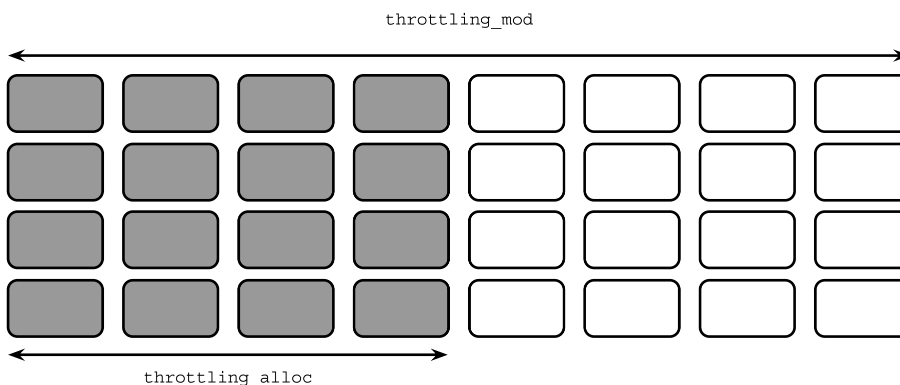


Figure 3.3: Example for throttling code

3.3 GPU Throughput Analysis

In this section we will figure out what feature affects GPU performance. To predict the optimal number of threads, we must know which factor affects on performance, so the application can be classified whether it is good for CPU or GPU. We know that locality is important for CPU performance, but we can not sure that same fact will be applied to GPU since it runs a large number of cores, so even if it has temporal locality, the line containing the data can be evicted

from GPU cache due to huge amount of memory access for all cores. Instead of using throttling kernel, we used unmodified code to detect naive performance of GPU which does not affected by throttling. You can find the description for each variable used in this section from Table 3.1. We used 4 kernels for Intra-analysis and 3 kernels for Inter-analysis. Table 3.2 shows the inner loop of each kernel.

Variable	**Description**
k	loop index
x	global_id(0)
y	global_id(1)
N	data size

Table 3.1: Kernel variable description

Kernel	**Intra**	**Inter**
1	B[x] = A[x][k] (N=16384)	C[y][x] = A[y][k]
2	B[x] = A[x][k] (N=25600)	C[y][x] = A[k][y]
3	B[x] = A[x][k] (N=28800)	C[y][x] = A[x][k]
4	B[x] = A[x][k] (N=16384, k<N)	

Table 3.2: Kernels for intra-thread analysis

3.3.1 Intra-Thread

We tested 1D, 2D GPU kernels with different amount of memory stride. Figure 3.4 shows how we make a memory access with different amount of stride.

We runs kernel with s range of 1, 2, 4, 8, 16. Kernel with $s = 1$ will access memory continuously, and others will have interval of s. The first three kernel are the same, but testes with different input sizes. In kernel 4, the inner loop line is same with other, but the loop variable k increase until N instead of $s \ N$

```
__kernel void micro_kernel(__global float *A,
                           __global float *B,
                           int N)
{
    int x = get_global_id(0);

    for(int k = 0; k < N; k+=1)
        B[x] = A[x*N + s * k];
}
```

Figure 3.4: Intra kernel 1 code

Figure 3.5 is graph of running time for five kernels. The time is normalized by time of stride 1. From kernel 1, we can see that performance is best on stride 4. However when we test larger data, on kernel 2, stride 2 perform best and on kernel 3 continuous version runs on shortest time. We can see that on small data size GPU internal cache affects on kernel. For example, on stride 2, since we change k from 0 to 2N-1, the memory region from a thread in which x is 0 collapsed with next thread (x=1) when k gets larger than N. When data gets larger this effect is lost due to limited cache size. On kernel 4, instead of increase k to $s \ N$, we only increase it to N and multiplied s to running time, to match with loop size. The result is similar to kernel 3, setting with stride 16 performs worst. With this results, we can see that we should consider locality also with

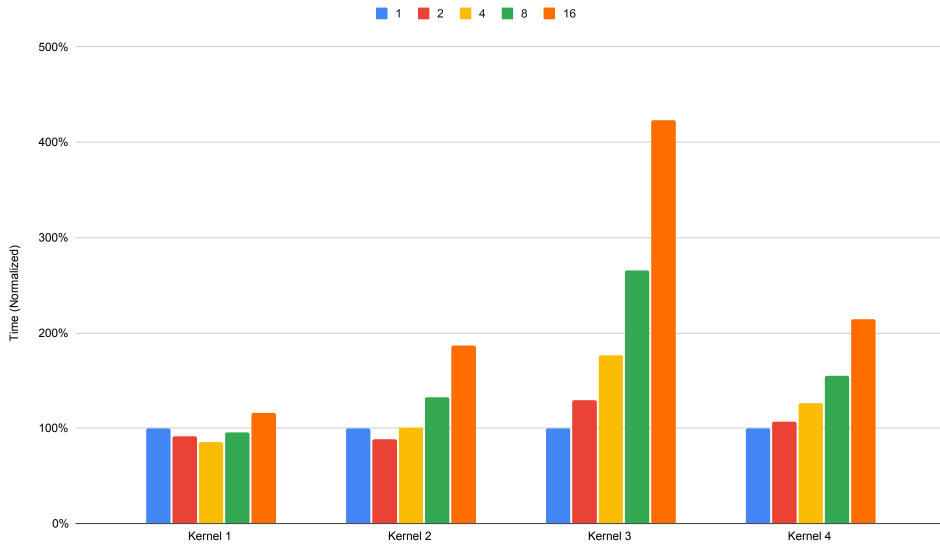


Figure 3.5: Normalized time for kernels with different amount of stride

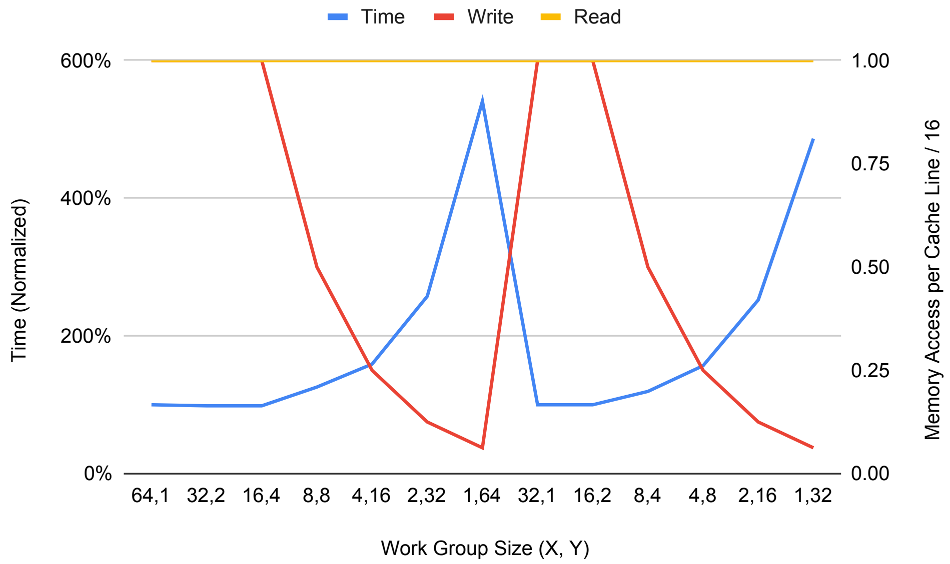
other thread's view.

3.3.2 Inter-Thread

The target kernel is 2D matrix copying. Figure 3.2 shows 8 similar but different kernels for Inter-work group analysis. We changed work group size from 64x1 to 1x64 so that variable x and y assigned differently but not affect to single thread's memory access pattern.

To analysis inter-thread memory access, we use calculate the number of memory accesses per cache line for same instruction. For example, if all 32 threads reference memory continuously, the number will be 16 since we use 4 byte floating point for the data. We represent cache efficiency by dividing the number by 16. We will call this simply efficiency in this section.

Figure 3.6 shows result of kernel 1 and 2 for different work group size. For



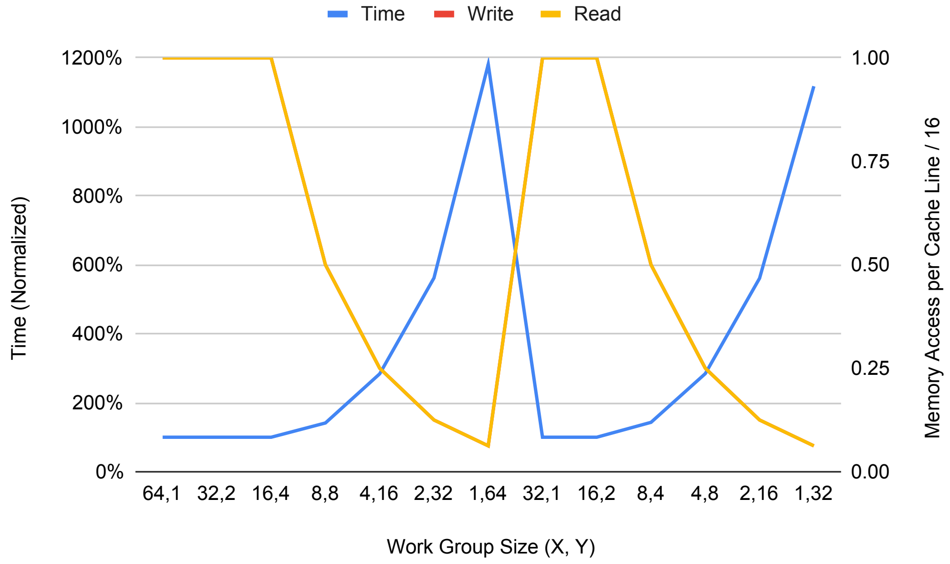


Figure 3.6: Normalized time for kernel 1, 2

kernel 1, read efficiency is 1 for all configuration. For kernel 2, it differs. On both kernel, we can see that when the write efficiency gets lower, the time it takes gets higher. Also for kernel 1, which read efficiency is always 100%, slowest configuration reaches about 600%, but for kernel 1, of which read efficiency also gets lower, slowest configuration reaches about 1200%.

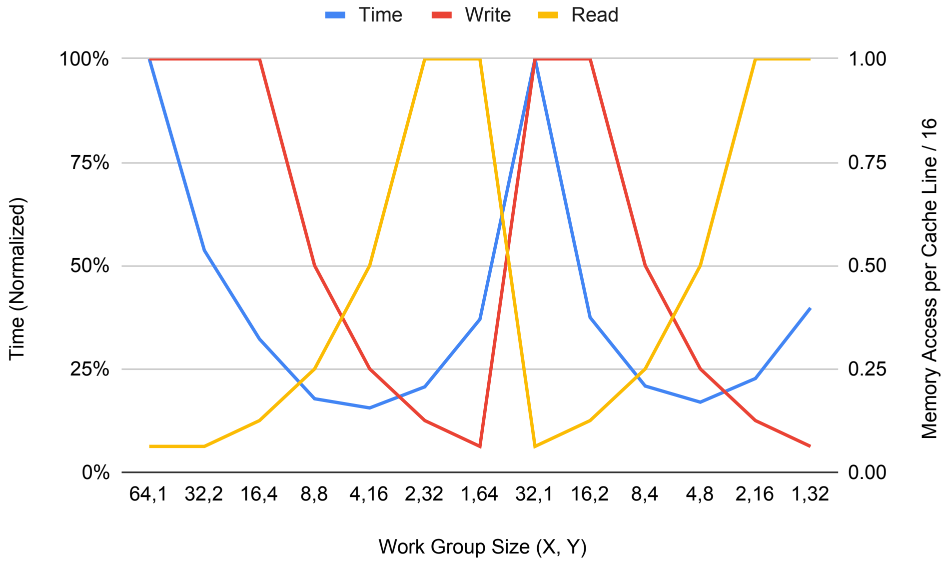


Figure 3.7: Normalized time for kernel 3

When efficiency for read and write differs, it show different results depends on how much they are low or high. In figure 3.6 when write efficiency goes down, read efficiency goes up. Time makes U shaped graph, results in minimum time near when the read and write crossed.

From these results, we can figure out what element affects on GPU performance. However the number of possible value is too much to design a single model, and also it will depends on GPU architecture. So we will use this information to make a machine learning model.

3.4 Prediction Model

3.4.1 Feature Selection

Using results from previous section, we designed model features to predict optimal number of cores to use. Each features are related to how many memory requests are produced from the device, which will affect optimal number of threads. Since our target is memory intensive application, we consider the computation ratio is low, and not include it.

Intra Thread Cache Efficiency From a thread's point of view, it accesses data sequentially in a loop. We find from Section 3.3.1 that the size of stride affects on GPU performance. We also know that it is important factor for CPU. To measure it, we calculate how much a single cache line is reused after it loads, and divide it by 16, which is the number of floating points one cache line can contains. We will call it as *Intra Thread Cache Efficiency*. We count lines only for 16 loops, which is the number of loop when continuous accesses fills one cache line. It can also applied for constant access to same memory inside a loop. If the kernel accesses same memory address for entire loop, the value will be 1.

Inter Thread Cache Efficiency This feature is more important to GPU performance. For each instruction, **T** (**T** can be different on architecture) threads will simultaneously make memory access (for some instructions or not). So we calculate efficiency by dividing *the number of cache lines **T** threads produces on a single line* by **T**.

Memory Access Pattern We also calculate basic memory access pattern of kernels. We tracked the number of memory accesses which is continuous,

constant, and have stride. The number of arithmetic calculation is also included.

Data Size From Section 3.3.1 we have seen that size of data affects with results. So we also put *data size* into features. It is determined by the maximum range of access occurs in a loop.

3.4.2 Model Training

Micro benchmark We made 1D and 2D kernels for training. For 1D kernel, we varies stride and used 1 and 2 matrices for read. For 2D kernel, we varies work group size to get different features. We have total 154 different kernels.

Model Design We made two models, blended by best three models from which show good prediction compare to others. For first model we use all data for training. We normalized time for each CPU, GPU configuration and for a workload, with minimum time of the workload results. We put core configuration as feature, so the best configuration in a workload will trains as a time 1.0, and others will be larger than that. On prediction, we put all possible core configuration in feature and pick the smallest results. We used regression model for this. We blended Light Gradient Boosting Machine, Random Forest Regressor, Extra Trees Regressor using Voting Regressor. The number of input data is multiplied by total core configuration, which is 17864 data on Intel, 12320 data on AMD.

For second model, we simplify the problem and train only best results. For each micro-kernels, find cpu and gpu combination which gives best (minimum) results, and train it as a class. Since we have to predict both CPU and GPU we have to combine these to one class to train the model. So we simplify the problem. We classify kernels to 4 class: 1) Using CPU 100% and some for

GPU is best (CPU) 2) Using GPU 100% and some for CPU is best (GPU) 3) Using both CPU and GPU 100% is best (ALL) 4) Using both CPU and GPU but both are not 100% is best (MIX). We blended Random Forest Classifier, Gradient Boosting Classifier and Extra Trees Classifier using Voting Classifier. The number of input data is same with the number of kernels.

Chapter 4

Experiment

4.1 Throughput Measurement

The main difficulty in measuring performance in integrated architecture is workload partitioning. Since if the CPU or GPU ends earlier than other it will be not accurate run-time of the kernel. Without modifying hardware architecture, after the kernel starts it is hard to add more workloads to kernel instead of relaunching it, so we have to estimate run-time of each device beforehand. As we want to focus on throughput of co-running applications, instead of measuring run-time of application, we measured time to process one work group.

If the number of total work groups are too small, measured throughput can be inaccurate. Figure 4.1 shows the problem. We track the time and the number of work groups until the kernel ends. However, if some CPUs failed to finish current work but almost done it, like on (A), and if the kernel ends after all CPUs finished their work, like in (B) they can make a big difference on performance even if its real performance is almost same. Assume the number of

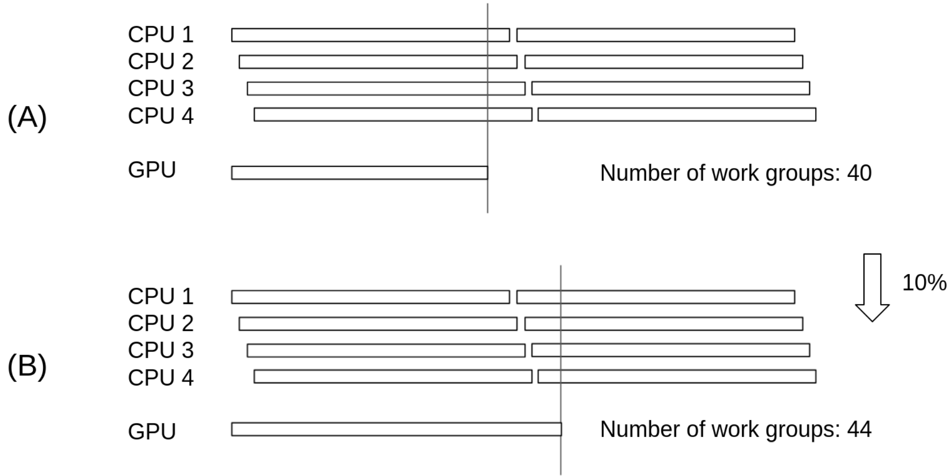


Figure 4.1: Performance with work group ending

processed work groups on (A) is 40, then the processed number of work groups on case (B) will be 10% more. To avoid this problem, we repeat the GPU kernel while the total number of the work group number become more than *the number of CPU cores * 100*. It makes that even if all CPU almost proceed the work group and not finished, the error for it will be less than 1%.

4.2 Target Device

We use AMD 3400G quad-core (eight threads with Simultaneous multithreading) APU (Picasso) system and a hexa-core (12 threads with Hyperthreading) Intel Comet Lake i5-10400 processor system. The AMD system integrates Vega 11 GPU, which has 11 CUs with each containing 64 Shaders. The Intel system contains an Ingel Gen9.5 UHD Graphics GPU which comprises 24 CUs. We use 64 for T on AMD system, however on the intel system, instead of cores there are registers and shared FPUs for CU. So we used 32, which is recommended

for work group size on opencl information for the system.

4.3 Micro-benchmark Results

Table 4.1: Results by Model-1

	Intel	AMD
ALL	138%	134%
Model	223%	170%

We trained model with 85% of micro-benchmark results and predict for remaining 15% of data.

Model-1 We compared average normalized time for prediction to CPU, GPU 100% case (ALL). Table 4.2 shows the results. The ALL method performs better on both system.

Table 4.2: Results by Model-2

	Intel	AMD
ALL	56%	17%
Model	86%	63%

Model-2 Table 4.2 shows the percentage of correct prediction for each labels on test set of microbenchmarks. When we use ALL methods, we can get limited number of correct prediction.

4.4 Real Benchmark Results

We have tested model on real benchmark. We used PolyBench[8] suite, since it contains linear algebra kernels which are memory bounded. The prediction of

model 1 for real workloads is not good. Table 4.3 shows the results. On both machine, it performs worse than ALL.

Table 4.3: Real Benchmarks Prediction Results - Model 1 (normalized time)

	CPU	GPU	ALL	Model
Intel	1.54	4.59	2.21	2.84
AMD	3.67	2.29	1.50	3.63

We also tested model 2 for real benchmark. Table 4.4 shows the results. We can see that model predicts almost 3 (ALL) for Intel, 1 (CPU Max) for AMD.

Table 4.4: Real Benchmarks Prediction Results - Model 2

	ALL	Model
Intel	138%	223%
AMD	134%	170%

4.5 Result Analysis

Even if we have tested the relativity of memory access patterns and performance with GPU, the model based on the result shows not good results. We analysis that the problem is related with feature design. We run DBSCAN clustering on the training data for model-2. As a results, we can see that data with different classes are belong to same cluster, which means that the feature of it are similar but classified different. It occurs mostly on class 1 (CPU Max) on Intel and 3 (ALL) on AMD. It means that we need more feature to separate those from other class.

For example, we added data size feature to a model, based on analysis of Section 3.3.1. However the main cause of different throughput is that the memory access of a thread collapsed with the different thread's access, but not

on same instruction. It means that we have to calculate inter-thread efficiency not only for simultaneous instruction but on instructions have some time gap, as the threads in work group uses same cache inside GPU CU.

Also as we calculate GPU features based on naive C kernel code, it can show different characteristics with real pattern. To analysis it more correct, we should get features using GPU assembly code.

Lastly, the modeling of training/test data can be a problem. For first model, we trains minimum case for all kernels as 1.0, treating the number of core as same feature as others. We can think that the model will groups all minimum cases from different kernels, which have different features onto same group with 1.0. Also for second model, we lost the application trend by training it with only best case. For example, there is a kernel which produces good results when using GPU 100% is good but the number of CPU does not affects. We but train this kernel class 3, which will be treated as same kernel with using both 100% is good.

Chapter 5

Conclusion

By doing GPU analysis, we can figure out that GPU performance can be bounded by limited memory bandwidth. We can throttle the number of cores modifying kernel code to execute for specific cores. By doing static analysis, we can predict the memory performance on each device, which can tell us whether the application is CPU friendly or GPU friendly.

Using the features extracted from the code and running configuration (data size, work group size), and using the micro-bench performance results for different core configuration from target device as a training data, we can build a model to predict the optimal core settings for CPU and GPU.

However the results of model is not good. To improve the results, we think that these things are needed: 1) Analysis on assembly code on CPU and GPU. 2) More features, especially on inter-thread in same work group but on different instructions.

Bibliography

[1] A. Branover, D. Foley, and M. Steinman. Amd fusion apu: Llano. *IEEE Micro*, 32(2):28–37, March 2012.

[2] Y. Cho, C. A. C. Guzman, and B. Egger. Maximizing system utilization via parallelism management for co-located parallel applications. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, pages 14:1–14:14, New York, NY, USA, 2018. ACM.

[3] Y. Cho, F. Negele, S. Park, B. Egger, and T. R. Gross. On-the-fly workload partitioning for integrated cpu/gpu architectures. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, pages 21:1–21:13, New York, NY, USA, 2018. ACM.

[4] M. Damschen, F. Mueller, and J. Henkel. Co-scheduling on fused cpu-gpu architectures with shared last level caches. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2337–2347, Nov 2018.

[5] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz. Inside 6th-generation intel

core: New microarchitecture code-named skylake. *IEEE Micro*, 37(2):52–62, Mar.-Apr. 2017.

[6] M. K. Emani and M. O'Boyle. Celebrating diversity: A mixture of experts approach for runtime mapping in dynamic environments. In *ACM SIGPLAN Notices*, volume 50, pages 499–508. ACM, 2015.

[7] J. Gómez-Luna, I. E. Hajj, L.-W. Chang, V. García-Floreszx, S. G. d. Gonzalo, T. B. Jablin, A. J. Peña, and W.-m. Hwu. Chai: Collaborative heterogeneous applications for integrated-architectures. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 43–54, April 2017.

[8] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10, May 2012.

[9] Intel. The compute architecture of intel processor graphics gen9. `https://software.intel.com/content/dam/develop/external/us/en/documents/the-compute-architecture-of-intel-processor-graphics-gen9-v1d0.pdf`, 2015.

[10] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali. Adaptive heterogeneous scheduling for integrated gpus. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 151–162, New York, NY, USA, 2014. ACM.

[11] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, Nov 2005.

[12] J. Lee, M. Samadi, Y. Park, and S. Mahlke. Skmd: Single kernel on multiple devices for transparent cpu-gpu collaboration. *ACM Transactions on Computer Systems*, 33(3):9:1–9:27, Aug. 2015.

[13] S. Mittal and J. S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4):69:1–69:35, July 2015.

[14] A. Raman, A. Zaks, J. W. Lee, and D. I. August. Parcae: A system for flexible parallel execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 133–144, New York, NY, USA, 2012. ACM.

[15] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli. Hetero-mark, a benchmark suite for cpu-gpu collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, Sept 2016.

[16] F. Zhang, B. Wu, J. Zhai, B. He, and W. Chen. Finepar: Irregularity-aware fine-grained workload partitioning on integrated architectures. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 27–38, Piscataway, NJ, USA, 2017. IEEE Press.

[17] F. Zhang, J. Zhai, B. He, S. Zhang, and W. Chen. Understanding co-running behaviors on integrated cpu/gpu architectures. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):905–918, March 2017.

[18] Q. Zhu, B. Wu, X. Shen, K. Shen, L. Shen, and Z. Wang. Understanding co-run performance on cpu-gpu integrated processors: observations, insights, directions. *Frontiers of Computer Science*, 11(1):130–146, Feb 2017.

요약

CPU와 GPU가 메모리를 공유하는 Integrated graphics(Intel) 또는 APU(AMD) 시스템에서 어플리케이션을 동시에 작업하는 것은 두 기기 간의 메모리 복사에 걸리는 시간을 절약할 수 있기 때문에 효율적으로 보인다. 한편으로 메모리 연산이 많은 비중을 차지하는 어플리케이션의 경우, 서로 다른 메모리 접근 특징을 가진 두 기기가 동시에 같은 메모리에 접근하면서 병목현상이 일어난다. 이 때 작업에 필요한 데이터를 기다리면서 코어가 중지되는 상황이 발생하며 퍼포먼스를 낮춘다.

이 논문에서 우리는 소프트웨어 코드의 변형을 통해 제한된 개수의 코어만을 사용하는 방법을 소개한다. 그리고 이 방법을 사용하여 코어 개수를 조절할 때 모든 코어를 사용하는 것보다 더 좋은 결과를 낼 수 있음을 보인다. 이 때 어플리케이션의 메모리 접근 패턴이 최적 코어 개수와 관련이 있음을 확인하고, 코드를 기반으로 해당 패턴을 분석한다.

퍼포먼스는 또한 시스템의 특징과도 관련이 있다. 따라서 우리는 가능한 메모리 패턴들을 담고 있는 마이크로 벤치마크를 생성하여 이를 시스템에서 테스트한 결과를 기계학습 모델의 학습에 사용하였다. 그러나 메모리 패턴의 특징이 충분하지 않아 모델은 좋은 예측 결과를 내지 못했다. 비록 모델 예측 결과가 좋지 못했지만, CPU-GPU 통합 아키텍처의 메모리 접근에 대한 분석으로서 의미가 있을 것이다.

주요어: 병렬성 관리, CPU-GPU 통합 아키텍처, 코드 분석

학번: 2019-22899