Performance Characterisation and Simulation of Intel's Integrated GPU Architecture

Prasun Gera*, Hyojong Kim*, Hyesoon Kim*, Sunpyo Hong†, Vinod George†‡, Chi-Keung (CK) Luk†‡
*Georgia Institute of Technology
{prasun.gera, hyojong.kim, hyesoon.kim}@gatech.edu
†Intel Corporation
sunpyo.hong@intel.com, vinodmgeorge@gmail.com, chikeung.luk@gmail.com

***Abstract*—Integrated GPUs (iGPUs) are ubiquitous in today's client devices such as laptops and desktops. Examples include Intel's HD or Iris Graphics and AMD's APUs. An iGPU resides on the same chip as the CPU, which is in contrast to a conventional discrete GPU that would typically be connected over the PCI-E bus. Much like discrete GPUs, iGPUs are also capable of general purpose computation in addition to traditional graphics roles. Further, iGPUs have some interesting differences compared to traditional GPUs such as a cache-coherent memory hierarchy and a shared last level cache with the CPU. Despite their wide spread use, they are not studied very extensively. To the best of our knowledge, this paper introduces the first open source trace generation and microarchitectural simulation framework for Intel's integrated GPUs. We characterise the performance of Intel's Skylake and Kabylake GPUs through detailed microbenchmarks, and use the performance evaluations to guide our models and validate the simulator.**

I. Introduction

Graphics Processing Units (GPUs) are widely used for graphics and general purpose (GPGPU) computation. Popularised by NVIDIA's CUDA framework for GPGPU programming, the last decade has seen a steady increase in use of GPUs for high performance computing and machine learning. While this growth has mostly been tied to conventional discrete GPUs in the past, a new class of GPUs, integrated GPUs (iGPUs), has become nearly ubiquitous in client devices such as laptops and desktops. iGPUs are a part of the same chip as the CPU. Due to the power, area, and thermal constraints, they are not designed to outperform their discrete counterparts in the raw compute throughput. However, compared to their discrete counterparts, which need to communicate over the PCI-E bus with the CPU, iGPUs have a much shorter path to the CPU. Further, recent iGPUs also feature a cache coherent memory hierarchy with the CPU. They share the last-level cache with the CPU and can access the large system DRAM, which enables optimisations such as zero-copy memory objects and pointer sharing. Vendor neutral frameworks such as OpenCL or OpenGL make it possible to port code written for discrete GPUs to iGPUs. These factors make iGPUs attractive candidates not only for light gaming or media workloads, but also for specific GPGPU tasks such as inference in machine learning, or offloading wide vector operations from the CPU. The energy efficiency of iGPUs is also seen as a strong point for future Internet of Things (IoT) devices.

We find that most of the research in the past, including tooling and infrastructure, has focused on discrete GPUs. NVIDIA GPUs have been studied extensively, and simulation or emulation frameworks such as GPGPU-Sim [1] and GPUOcelot [2] have been used for GPGPU research. While there are similarities between discrete and iGPUs, no public simulator exists for simulating workloads for Intel iGPUs. To that end, this paper makes the following contributions:

- We characterise the performance of Intel's Skylake and Kabylake iGPUs through a collection of microbenchmarks. In particular, we study the memory characteristics in detail, and also cover floating point performance and the thread scheduler's behaviour.
- To the best of our knowledge, we introduce the first instruction level trace generation framework for Intel iGPUs built upon the GT-Pin [3] toolkit. The generated traces use a portable, simulator agnostic format serialised as Google Protocol Buffer [4] messages to encourage interoperability across simulators.
- We develop the iGPU module in MacSim [5], an open source heterogeneous architecture simulator. We use the performance characteristics collected earlier to model and validate the simulator.

II. Background

A. Intel's GPU architecture

We summarise the key characteristics of Intel's Gen9 iGPU architecture [6], [7] in Fig. 1. The GPU contains a large number of execution units (EUs) that perform SIMD computation. A collection of 8 EUs form a subslice. The subslice also contains a common instruction cache, and L1 and L2 sampler caches. Each subslice also contains a memory load/store unit called the data port. Three subslices are aggregated into one slice. The slice additionally consists of the L3 data cache, and a highly banked shared local memory (SLM). The L3 cache is a general purpose cache that is a part of the overall coherence domain with the CPU, whereas the L1 & L2 caches and SLM are local to the slice, and are not in the coherence

†‡ At Intel Corporation at the time of writing

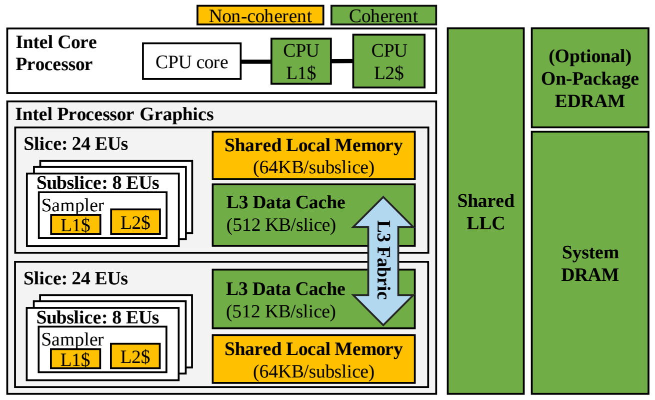
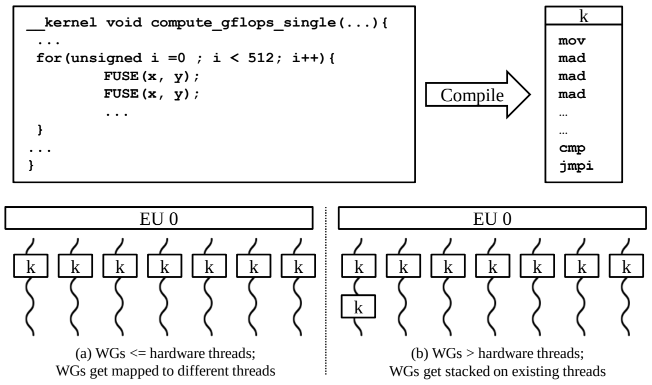


Fig. 1: Intel's Gen9 iGPU architecture


```
__kernel void compute_gflops_single(...){
 ...
 for(unsigned i =0 ; i < 512; i++){
        FUSE(x, y);
        FUSE(x, y);
        ...
 }
...
}
```
```
k
mov
mad
mad
mad
...
...
cmp
jmpi
```

(a) WGs <= hardware threads; WGs get mapped to different threads

(b) WGs > hardware threads; WGs get stacked on existing threads

Fig. 2: iGPU scheduler

domain. Some products have more than one slice. In such designs, multiple slices are connected via an L3 interconnect fabric which combines the total L3 capacity across the slices and presents the L3 as a logically monolithic cache. The GPU and the CPU connect to the shared last level cache (LLC) via a ring interconnect. The LLC would then connect to an on-package eDRAM cache if the product has one. The eDRAM cache would finally connect to the system DRAM, or the LLC would connect to system DRAM if there is no eDRAM cache.

B. OpenCL

OpenCL is a programming framework used for parallel and high performance computing with wide device support across CPUs and GPUs. OpenCL uses a notion of work groups and work items to express parallelism. A work group consists of several work items that execute in parallel, and the application kernel consists of several work groups that execute in parallel. Work items within a work group have access to local state and resources in the form of synchronisation primitives and shared local memory (SLM). There are some noticeable differences between Intel's OpenCL implementation for iGPUs and NVIDIA's CUDA implementation for discrete GPUs. In CUDA terminology, thread blocks, or cooperative thread arrays (CTAs), are roughly equivalent to OpenCL work groups. CTAs are assigned to streaming multiprocessors (SMs) based on the availability of resources such as registers and shared memory. Each CTA is further divided into groups of 32 threads, known as warps. All the threads within a warp typically share the same instruction stream. A warp, which is a collection of scalar threads, is the basic unit that runs on an SM. The CTA scheduler is responsible for the fast context switching of warps. The scheduler can pick ready warps from the same or different CTAs. Note that all the threads described so far for CUDA are software threads. In contrast, Intel's GEN architecture has a well defined notion of hardware threads. Hardware threads in this context represent a unit of Simultaneous Multithreading (SMT) scheduling. *i.e.*, Each EU is able to simultaneously run 7 hardware threads in a GEN9 GPU. The global thread dispatcher is responsible for load balancing thread distribution across the entire device [6]. Further, each thread runs Single-Instruction-Multiple-Data (SIMD) instructions with support for predication and branching. From our experiments, we found that:

- Each OpenCL work group is mapped to a new hardware thread, up to the maximum supported hardware threads.
- A work group may also be split and mapped to multiple hardware threads. The main constraint in splitting work groups is the presence of barriers and shared memory, which limit where work groups get dispatched.
- If there are more work groups than the maximum supported hardware threads, the kernel invocations for the remaining work groups are stacked at the end of existing hardware threads (Fig. 2).
- Inside a thread, a kernel consists of series of variable width SIMD instructions. For instance, 16 OpenCL work items can be mapped to each lane of SIMD-16 instruction. [*]

C. GT-Pin

GT-Pin [3] is a binary instrumentation framework for Intel GPUs. It serves a similar purpose as Pin [8] for CPU workloads in that it can be used as the foundation for tasks like trace generation or other forms of static and dynamic analysis. We built our trace generation framework on top of the APIs provided by GT-Pin for the following reasons:

- Trace generation should be agnostic to the OS. GT-Pin has been tested on GNU/Linux and Windows.
- Trace generation should be agnostic to the programming framework. Since GT-Pin works at the native binary level, it can, in principle, be used for any GPU framework. This will likely be an important factor in the future since there are some proposals for convergence of OpenCL and Vulkan APIs [9].
- Trace generation should not require the source code of application or libraries. GT-Pin can instrument binaries at the native ISA layer, as opposed to an intermediate ISA layer such as PTX for NVIDIA. PTX based simulation frameworks such as GPGPU-Sim [1] and GPUOcelot [2]

[*]At a high level, a CUDA warp can be viewed as a fixed 32-lane wide instruction while in Intel GPU, the length is variable but typically 16.

TABLE I: CPU Parameters

Parameter	i7-6700k (SKL)	i7-7567U (KBL)
Cores	4	2
Threads	8	4
Base Freq	4.00 GHz	3.50 GHz
Max Turbo Freq	4.20 GHz	4.00 GHz
L1 I-cache (per core)	32 KB	32 KB
L1 D-cache (per core)	32 KB	32 KB
L2 cache (per core)	256 KB	256 KB

TABLE II: iGPU Parameters

Parameter	HD 530 (SKL)	Iris Plus 650 (KBL)
EUs	24	48
H/W Threads per EU	7	7
SIMD FP Units per EU	2	2
SIMD FP Width per unit	32 bits	32 bits
Base Freq	350 MHz	300 MHz
Max Dynamic Freq	1.15 GHz	1.15 GHz
Number of Slices	1	2
Number of Sub-Slices	3	6
L3 data cache	512 KB	1 MB
SLM (per sub-slice)	64 KB	64 KB

TABLE III: Shared Resources

Parameter	i7-6700k + HD 530 (SKL)	i7-7567U + Iris Plus 650 (KBL)
LLC	8 MB	4 MB
eDRAM	N/A	64 MB
DRAM	DDR4 2133 MHz	DDR4 2133 MHz

need the source code of applications to produce PTX output, which is not always available for a lot of libraries. This makes simulation of modern workloads challenging. While SASSI [10] enables instrumentation of NVIDIA's native SASS ISA, it still needs PTX as a starting point.

III. Performance Modelling

We use a collection of microbenchmarks written in OpenCL to understand the performance characteristics of Skylake (SKL) and Kabylake (KBL) iGPU architectures. The specifications for the hardware used for these experiments are listed in Tables I, II & III. There are not too many significant differences micro-architecturally between Skylake and Kabylake. However, the two processors that we used are of different types, one being a desktop processor (i7-6700k (SKL)), and the other being a mobile processor (i7-7567U (KBL)) from a Next Unit of Computing (NUC) device. That makes the comparison interesting nonetheless since the two processors allocate their power, thermal and area budgets differently. The mobile KBL processor trades off CPU resources for GPU resources since its primary use is media centred. The KBL processor has 2 cores as opposed to 4 in SKL and lower CPU clocks, but it has twice the number of EUs in its GPU and additional eDRAM cache. We contrast the results between them in our experiments when the comparison is resource oriented. When the trends are similar, we focus only on the SKL desktop processor for brevity.

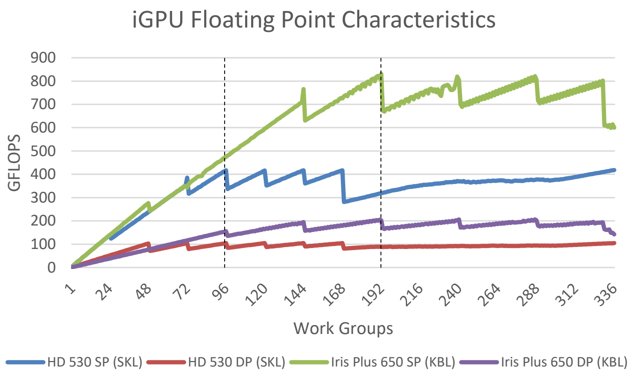


Fig. 3: Each Work Group is comprised of 32 Work Items. SP = Single Precision, DP = Double Precision. Vertical dotted lines represent the minimal work groups that achieve peak throughput for the two GPUs

A. *Floating Point Performance*

In this microbenchmark, we use fused multiple and add (MAD) instructions to characterise the floating point performance of the GPUs. Each work group (WG) is comprised of 32 work items (WIs), and we vary the number of WGs, while measuring the throughput as Giga floating point operations per second (GFLOPs). We found empirically that 32 WIs per WG were sufficient to reach close to the peak throughput. At lower WIs, some instructions get masked at runtime, which would need more WGs to saturate throughput. The results of this experiment are presented in Fig. 3. The two dotted vertical lines represent the minimal number of work groups, 96 and 192, for which the hardware achieves peak GFLOPs for the two GPUs.

Each EU can support a maximum of 7 hardware threads, and we have either 24 or 48 EUs for our SKL and KBL GPUs respectively. Hence, the GPUs can support a maximum of 168 or 336 hardware threads respectively. We notice that the performance scales pretty linearly in the beginning as we increase the work groups, which suggests a round robin scheduler between the EUs. We also see periodic dips in throughput. This is due to the workload imbalance across EUs for certain WG configurations. There is an imbalance if the number of active threads across EUs are not uniform, or if the number of WGs is greater than the number of available EUs, in which case we see the stacking effect as shown earlier in Fig. 2. When the workload distribution is not uniform, we can end up with a subset of EUs being on the critical path, and the rest being idle. This leads to an overall drop in throughput as evidenced by the dips in Fig. 3.

The peak theoretical GFLOPS for the hardware can be calculated as follows:

$$Max\ GFLOPS = (EUs) \times (SIMD\ units/EU) \times (FLOPs\ per\ cycle/SIMD\ unit) \times (Freq\ GHz) \quad (1)$$

Based on the Intel's documentation [6], the hardware details

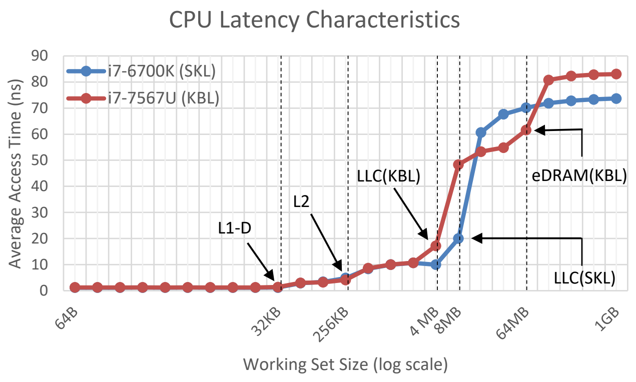


Fig. 4: Average access time per access for CPU

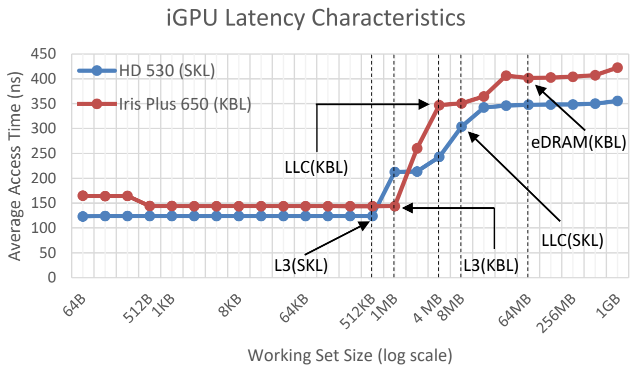


Fig. 5: Average access time per access for iGPU

for floating point computation are as follows: Each SIMD unit is 4-wide with a peak issue rate of 1 MAD instruction per cycle. The SIMD units are fully pipelined. Each EU has 2 such SIMD units. Each EU can issue instructions from 4 threads every cycle. The double precision (DP) throughput of each SIMD unit is $\frac{1}{4}$(single precision (SP) throughput). The floating point latency is 4 cycles. A 4-wide MAD operation translates to 8 operations. Our GPUs have either 24 or 48 EUs, and a peak frequency of 1.15 GHz. Plugging in these values in equation 1, we get $24 \times 2 \times 8 \times 1.15 = 441.6$ GFLOPs as the peak SP throughput for the SKL GPU, and 883.2 GFLOPs for KBL due to twice the number of EUs. The corresponding DP throughput is scaled down by a factor of 4, which leads to 110.4 and 220.8 DP GFLOPs respectively. Further, we also notice that 4 hardware threads per EU are sufficient to hide the floating point latency of 4 cycles and achieve close to the peak throughput. At 96 or 192 work groups for the two GPUs, we are at an occupancy of 4 threads per EU in a fair round robin schedule, which achieves pretty close to the peak GFLOPs that we calculated earlier. This is marked by the dotted vertical lines in Fig. 3. Going beyond that occupancy does not benefit the throughput.

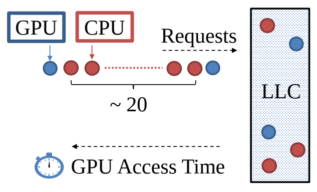
B. Memory Hierarchy Latency

We use a single threaded random access microbenchmark to determine the latencies for various levels of the memory hierarchy. Since the accesses are dependent and random, it reduces the impact of prefetchers or other optimisation techniques. We vary the working set size for the microbenchmark, and observe the average access time for each working set size. It resembles a pointer chasing workload, although we use an array instead of raw pointers in our code. We create a random cycle of offsets in the input array such that each access is cacheline aligned. The pseudo-code for the microbenchmark is:

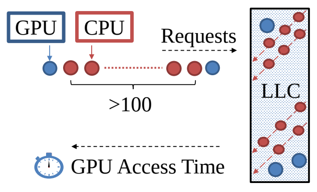
```
initialize (input_array);
result = 0;
for(i= 0; i < n; i++){
        result = input_array[result];
}
```

Listing 1: Cache latency microbenchmark

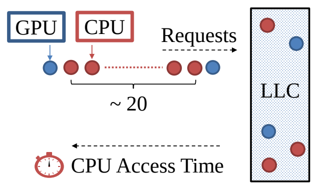
Figs. 4 and 5 show the average access time at different working set sizes for both SKL and KBL CPUs as well as GPUs. Throughout this section, access time refers to the total time including the miss time from higher levels of the hierarchy. We notice that the access times for the entire memory hierarchy are generally higher for the iGPUs compared to the CPUs. The GPUs have a much lower frequency than the CPUs. For instance, the SKL GPU has a base frequency of 350 MHz and maximum frequency of 1.15 GHz, compared to the base frequency of 4.00 GHz and maximum of 4.20 GHz for the CPU. Hence, the latency gap in terms of native cycles is lower than the difference in absolute time. The first general purpose cache in these iGPUs is the L3 cache. L1 and L2 are specialized sampler caches which are not accessed in this benchmark. The latency for L3 accesses from GPUs is about 125 ns for SKL, and 144 ns for KBL. The total L3 capacity for KBL is 1 MB since it has two slices. Next in the hierarchy is the last level cache (LLC), which is also shared with the CPU. This also highlights the difference between the latency of accessing the same resource between the CPU and the GPU. The access time for the LLC from the CPU is $\sim$ 10 ns, whereas from the GPUs, it starts at 212 ns for SKL and 260 ns for KBL. Further, the access time steadily increases in the LLC region for the GPUs. The latency curves in the 1 MB - 8 MB range and 2 MB - 4 MB range for the two GPUs show increasing access times (Fig. 5) although one would expect the working set to fit in the LLC. In contrast, the CPU's curves are pretty flat in the LLC region (Fig. 4). This leads us to believe that the GPU is not able to take advantage of the full capacity of the LLC, and perhaps some capacity is always reserved for the CPU. Going further down the hierarchy, we notice that, for KBL, the access time is tempered by the eDRAM cache in the 4 MB - 64 MB region. The effect on latency is more pronounced for the CPU than the GPU, where we see the KBL CPU's curve fall below the SKL curve in the eDRAM region (Fig. 4). Similar to the LLC, there is a disparity between access times from the CPU and GPU to the eDRAM cache, and the GPU's access time increases noticeably even in the eDRAM range. The access time for the eDRAM cache from



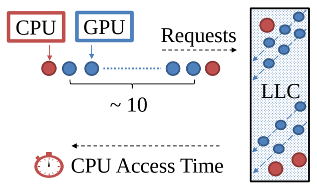
(a) 20 intervening CPU requests b/w GPU requests



(b) More than 100 intervening CPU requests b/w GPU requests



(c) 20 intervening CPU requests b/w GPU requests



(d) 10 intervening GPU requests b/w CPU requests

Fig. 6: Experimental Setup for Shared LLC interference experiments. Results in Fig. 7

the KBL CPU starts at $\sim$ 48 ns, whereas for the GPU, it starts at $\sim$ 350 ns. Finally, for 128 MB and beyond, we observe stable average DRAM access times of about 73 ns and 83 ns for SKL and KBL CPUs, and 355 ns and 422 ns for the respective GPUs. In addition to lower core frequency, the higher access times for the GPU could also be a result of higher interconnection network latency. It is worth noting that this sort of a single threaded pointer chasing workload is unusual for a GPU. GPUs are designed for high throughput, and not low latency.

C. *Cache sharing effects between the CPU & GPU*

As we saw in subsection III-B, accesses to the shared LLC show different characteristics for the CPU and the GPU. We try to understand the effects of potential interference that either of them may cause on the other. For the next experiment, we run the benchmark on the CPU and the GPU concurrently, and measure the latency from one while we vary the interference from the other. We also vary each agent's working set to span sizes that cover the LLC. We use the SKL GPU for these experiments, and vary the working set from 1–9 MB for each agent to isolate the effects caused by the shared LLC. Fig. 6 describes the experimental setup. The first scenario that we evaluate, as depicted in Fig. 6(a), is one in which both the CPU and GPU use a similar single threaded pointer chasing workload as III-B, and we measure the GPU's average access time per request. We also repeat the same setup in Fig. 6(c), with the difference that we measure the CPU's average access time instead of the GPU's. In both these cases, more than 20 CPU requests arrive at the LLC for every request from the GPU due to the difference in raw latencies. Since we use single threaded dependent accesses for both the agents, this difference in request rates is determined entirely by the difference in latencies. We also evaluate two more scenarios. In Fig. 6(b), we measure the GPU's access time using the same method, but increase the interference from the CPU by using a multi-threaded streaming workload. The difference in rate of requests is even more pronounced than Fig. 6(a), and now more than 100 CPU requests arrive between subsequent GPU requests. Finally, in Fig. 6(d), we measure the CPU's average acess time using single threaded dependent accesses, but the GPU streams data using multiple threads. The GPU is able to stream out one cacheline every cycle, which translates to a rate of roughly 1 cacheline every 1 ns. Compared to the CPU's access time of roughly 10 ns, we see that the GPU's requests arrive at a rate roughly 10 times higher than the CPU in this scenario.

We present the results of these four experiments in Fig. 7. We make the following observations from the results:

- In Figs. 7(a) and 7(b), we see that,
  - The GPU's access times in the top left quadrant are similar in both the figures.
  - The GPU's access time does not increase appreciably along the horizontal axis in the top left quadrants.
  - The GPU's access time is slightly higher in the first column than the second column.
  - In Fig. 7(b), the GPU's access time ramps up significantly in the last few columns.
- In Fig. 7(c) we see that the CPU's access time is largely unaffected except in the last two rows.
- In Fig. 7(d), we see a gradient of increasing access times as we go down along the diagonal.

When the total working set fits in the LLC, the GPU's access time seems relatively unaffected by the nature of CPU's interference. We draw two conclusions from this. One is that the critical path for the GPU to access the LLC lies elsewhere, and is independent of the CPU's activity. The GPU's access time increases more because of an increase in its own working set than any sort of interference. This is related to our earlier point in III-B that the GPU is not able to take advantage of the entire capacity of the LLC. Secondly, the CPU is not given any preferential treatment in accessing the LLC. Despite the barrage of requests from the CPU in Fig. 7(b), we do not see any undue starvation at the GPU's end. We do see increased contention at DRAM though from CPU's streaming interference, as evidenced by higher values in the last couple of columns. Here, the CPU is streaming from system DRAM, and the GPU's average access time is dominated by the contention at DRAM. A slightly surprising result in this data is that the GPU's access time without any interference (first column Fig. 7(a), 7(b)) is actually a bit higher than the time with some interference (second column). We think that this may be a side-effect of frequency scaling or power management, which causes the GPU alone to run at a slightly lower performance when the CPU is idle compared to the case when both CPU and GPU are active. Moving on to the CPU's access time, Fig. 7(c) shows that the CPU is largely unaffected when the GPU's interference is single threaded. Since the GPU's access frequency is 1 in 20 relative to the

GPU Working Set	(Pointer Chasing Interference) CPU Working Set									
	0 MB	1 MB	2 MB	3 MB	4 MB	5 MB	6 MB	7 MB	8 MB	9 MB
1 MB	213.54	206.77	207.03	208.34	213.66	216.56	228.67	251.68	217.65	210.73
2 MB	215.01	209.33	210.31	215.04	219.57	236.35	249.27	269.91	278.53	275.69
3 MB	231.21	224.12	226.19	234.33	252.32	267.49	280.07	292.09	293.06	293.18
4 MB	246.6	240.19	247.03	262.01	275.24	290.32	298.37	302.39	302.1	302.45
5 MB	258.73	258.04	271.67	281.84	294.99	301.75	307.4	308.22	308.21	308.26
6 MB	273.68	270.74	290.09	296.31	305.99	309.7	312.04	312.57	311.94	312.21
7 MB	286.89	287.13	301.84	305.76	311.56	315	315.53	315.36	314.85	315.09
8 MB	304.29	303.37	310.58	314.81	315.68	317.36	317.55	317.49	316.79	317.07
9 MB	318.28	312.89	314.9	318.84	318.48	319.44	319.3	319.21	318.53	318.65

(a) ST Interference from the CPU. Measured = GPU access time

GPU Working Set	(Streaming Interference) CPU Working Set									
	0 MB	1 MB	2 MB	3 MB	4 MB	5 MB	6 MB	7 MB	8 MB	9 MB
1 MB	213.54	207.64	209.24	209.84	215.48	226.19	234.33	286.03	320.9	339.46
2 MB	215.01	208.24	210.82	217.44	227.23	238.55	257.8	305.11	342.82	355.22
3 MB	231.21	224.43	238.79	244.45	254.61	273.66	288.92	343.72	378.46	384.57
4 MB	246.6	236.95	256.38	263.73	284.87	291.62	304.93	364.87	392.38	393.8
5 MB	258.73	260.2	273.42	285.55	298.02	304.65	312.19	368.42	404.14	412.82
6 MB	273.68	278.19	289.93	299.94	310.77	312.49	316.84	381.05	416.06	411.42
7 MB	286.89	293.29	304.67	312.66	318.3	319	319.83	387.87	417.56	424.25
8 MB	304.29	302.3	316.44	319.69	320.6	321.25	322	389.3	419.35	428.99
9 MB	318.28	312.4	321.94	322.53	322.88	323.53	323.86	324.09	437.51	440.37

(b) STRM Interference from the CPU. Measured = GPU access time

CPU Working Set	(Pointer Chasing Interference) GPU Working Set									
	0 MB	1 MB	2 MB	3 MB	4 MB	5 MB	6 MB	7 MB	8 MB	9 MB
1 MB	9.8	10	9.8	9.75	9.8	10	9.96	9.88	9.76	9.75
2 MB	10.91	10.74	10.45	10.48	10.45	10.76	10.69	9.44	9.21	10.49
3 MB	9.23	10.97	10.71	10.7	10.71	9.51	9.51	9.61	9.24	9.25
4 MB	9.75	10.04	9.88	9.98	9.88	10.04	9.7	9.39	9.12	9.45
5 MB	9.97	11.51	10.39	10.1	10.39	10.66	9.45	9.81	9.94	9.61
6 MB	10.36	10.67	11.61	11.69	11.61	10.5	10.72	9.26	9.49	10.05
7 MB	11.69	21.74	16.83	17.19	16.83	17.15	13.24	12.89	11.25	13.3
8 MB	20.58	35.26	35.27	35.23	35.27	35.51	33.72	34.63	33.61	32.78
9 MB	31.96	45.67	50.19	48.85	50.19	50.01	49.55	51.38	50.29	49.71

(c) ST Interference from the GPU. Measured = CPU access time

CPU Working Set	(Streaming Interference) GPU Working Set									
	0 MB	1 MB	2 MB	3 MB	4 MB	5 MB	6 MB	7 MB	8 MB	9 MB
1 MB	9.8	10.42	10.22	10.74	12.78	18.5	29.86	42.14	67.07	88.09
2 MB	10.91	10.9	11.92	11.02	18.02	23.95	38.86	58.51	78.91	98.46
3 MB	9.23	9.64	9.78	10.96	25.63	34.37	46.18	64.34	80.9	96.3
4 MB	9.75	10.11	12.16	19.26	27.29	41.39	53.71	68.46	82.62	98.4
5 MB	9.97	10.16	16.69	31.38	39.66	50.23	61.39	72.67	82.22	98.68
6 MB	10.36	12.56	27.17	40.22	49.06	56.91	65.35	75.33	83.11	99.48
7 MB	11.69	18.37	35.44	48.84	55.02	60.58	68.94	77.9	83.91	99.64
8 MB	20.58	35.07	46.75	54.51	58.11	62.9	70.37	78.58	83.37	99.71
9 MB	31.96	46.48	53.45	59.1	60.52	64.61	71.18	79.67	83.97	99.59

(d) STRM Interference from the GPU. Measured = CPU access time

Fig. 7: Effect of CPU and GPU's interference on the access time to the LLC for SKL (8 MB LLC). Time measured in ns. ST = Single-threaded pointer chasing kernel; STRM = Multi-threaded streaming kernel. Measurement uses the ST kernel; interference is ST or STRM

CPU, this is not very surprising. The only increase that we see in the last two rows is again a result of spilling outside the LLC to DRAM, and is a result of the CPU's own working set. Finally, in Fig. 7(d), the GPU is streaming data, and sending requests at a 10 to 1 ratio compared to the CPU. Here, we see that the CPU's access times are affected even when the total working set is inside the LLC. For instance, with CPU's working set as 5 MB and GPU's as 2 MB, the access time is 16.69 ns (3rd column) whereas without any interference, it is 9.97 ns (first column). This leads us to believe that GPU is causing eviction of CPU's lines. There is increased contention at the DRAM level too, as seen by the steady increase of access times outside the LLC.

D. Memory Bandwdith and Coalescing

We study the behaviour of GPU's global memory, which is also the system DRAM for Intel iGPUs, in this section. Both our systems have dual channel DDR4 2133 MHz memory, and we discuss our observations from experiments on the SKL system. Memory coalescing is a well studied phenomenon in GPU architectures, and it is an important optimisation technique in high performance computing (HPC) applications. Memory coalescing refers to combining multiple memory accesses into a single transaction. In this experiment, we vary the number of work groups, the number of work items, and a stride parameter to cover a wide variety of access patterns. Fig. 8 describes our experimental setup. The total memory region is divided between work groups with no overlap. Within each work group, each work item reads a fixed number of words, 256 in our case. No address is accessed more than once. This is to ensure that we do not see any cache effects, and only characterise DRAM's performance. For stride = 1, consecutive work items in a work group access consecutive words. These would be coalesced into a single memory request when sent to DRAM. For stride = 2, the accesses are spaced out by one word between consecutive work items. At stride = 16, each work item would read from a new cacheline. In all these experiments, the total number of words accessed is the same. We measure the raw memory bandwidth based on the number of cachelines read from DRAM. We present the results in Fig. 9. At the bottom left corner of every sub-figure, we have a single work group with 16 work items. This translates to a single hardware thread, which is not sufficient to saturate

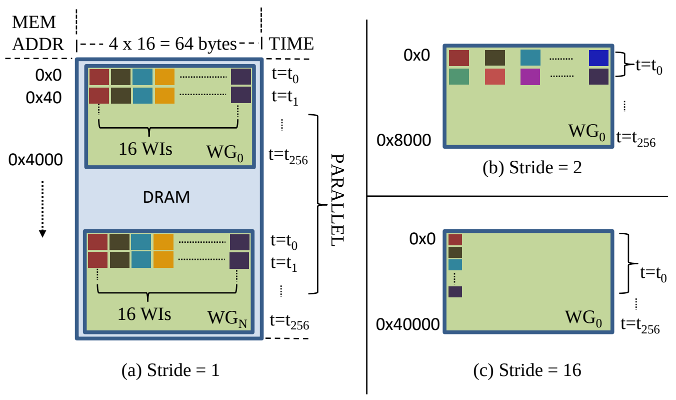


Fig. 8: Memory Coalescing microbenchmark. Example with 16 work items (WIs) per work group (WG). Each row is a cacheline (CL). For stride 1, 16 WIs access 16 words in a CL. For stride 16, 16 WIs access 16 consecutive CLs. Results in Fig. 9

the memory bandwidth, even at stride 16. As we increase the number of work groups and work items, we create more memory level parallelism (MLP) by generating more requests. Increasing the stride also increases the number of requests generated. As we move up along the main diagonal, we see an increase in bandwidth. We reach close to the peak bandwidth a lot sooner for larger strides. This is expected since we generate sufficient memory requests without relying on a large number of hardware threads.

E. Memory Level Parallelism

We run some further experiments to characterise the effect of memory level parallelism (MLP). The goal for this micro-benchmark is to determine the number of in-flight memory requests that can be serviced concurrently. Latency for memory requests can be hidden by multiple outstanding requests, also known as the MLP. If the system has an MLP of K, majority of the memory service time for up to K in-flight requests can be hidden. However, if the memory pressure increases beyond the available MLP, we would see the execution time increase. To measure this empirically, we use work groups with a single work item, and increase the work groups one at a time. Each work item's accesses are dependent, but they are independent from other work groups. We vary two parameters in this experiment: the number of work groups and the working set size of each work group; *i.e.*, we run Nx1 kernels where N is the number of work groups, and 1 denotes the single work item for each work group. Each work group translates to a hardware thread, and we vary N up to 168 since the SKL GPU supports a maximum of 168 threads. Since we run N work groups, the total working set size for the kernel is N times the working set size of a single work group. We choose different working sets per work group such that the total working set would fall at different levels of the memory hierarchy. As we increase the number of work groups, the total work done also increases proportionally.

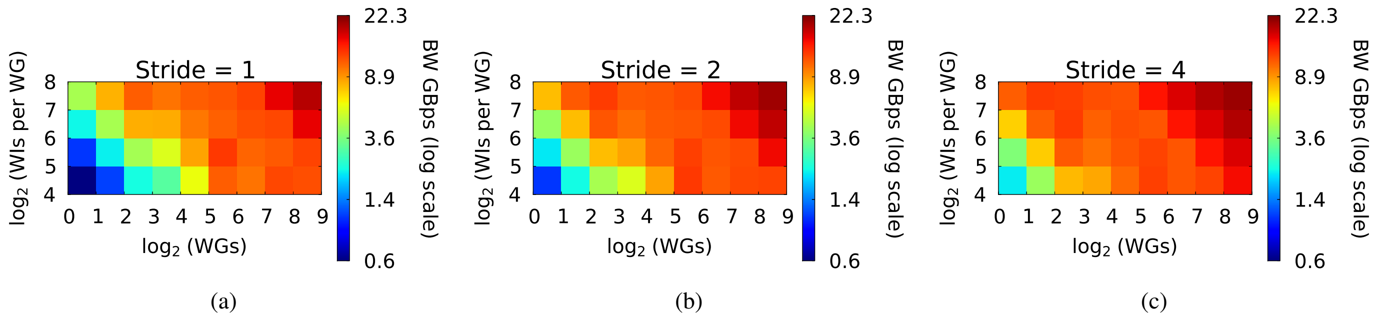
Fig. 10 shows the results for this microbenchmark. When the working set is 64 bytes per work group, or 2 KB per work group, the maximum total working set is smaller than the L3 cache. Hence, we see no difference in the performance characteristics of those two cases. At 1 MB per work group, the total working set starts off in the LLC, but soon exceeds the LLC around the 8 work group point. This is why there is an initial ramp in the relative time for the 1 MB case, after which it coincides with the 8 MB case. At 8 MB per work group, we already start outside the LLC, and observe the DRAM's effects. We see that for all the working set values, there is almost no increase in the total time up to 100 work groups (discounting the initial ramp for the 1 MB case). Note that the total work done and the amount of data read is increasing proportionally in this region. After this point, we start seeing an increase in the total time. The L3's latency is $\sim$ 125 ns, and throughput is 1 line per cycle. Hence, it makes sense that we are able to hide the work of more than 100 work groups behind the L3 latency. Interestingly, this trend persists even after we are outside the L3, as it can be seen for larger working sets. Since higher levels of the hierarchy have a higher latency, one would expect to hide even more work behind the latency. However, it appears that we hit a natural limit around the 100–120 work groups region irrespective of the total working set size. We believe that miss status holding registers (MSHRs) or other similar bookkeeping structures impose a limit on the available MLP. We conclude that the maximum avaliable MLP for the iGPU is close to 100.

IV. Trace Generation

We build the trace generator using the GT-Pin binary instrumentaion toolkit, which also includes the Gen encoder decoder (GED). These are loosely equivalent to Pin [8] and XED toolkits for x86 CPUs. GT-Pin is capable of registering callbacks at different granularities [3] such as function calls, basic blocks, and instructions. Our main objective is to collect complete execution traces including memory accesses. Fig. 11 presents a high level overview of the trace generation process. Due to the high performance overhead of frequent callbacks, and a few technical limitations, we limit our callbacks to the following events:

- Every basic block (BBL) that does not change the control flow of the program (non-CFG changing BBL)
- Memory accesses

The decision to instrument only non-CFG changing BBLs is internal to GTPin's algorithm. We infer the missing basic blocks during post processing from the control flow graph. The trace collection proceeds along two separate paths for basic blocks and memory accesses which are combined in the end during post processing. We use Google Protocol Buffers [4] as a serialisation format for the traces to encourage interoperability and reuse of traces across simulators. Another reason for this choice is to accommodate future versions or extensions to the ISA without breaking existing traces. Reading of traces is mostly done by the standardised ProtoBuf abstraction layer. The traces have a natural hierarchy that can



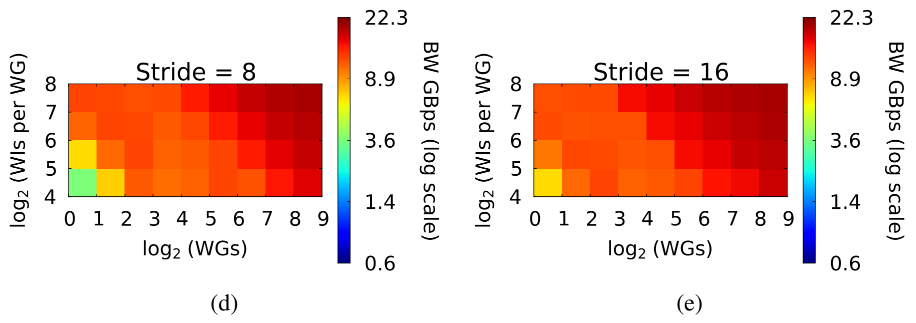


Fig. 9: Raw memory bandwidth for independent memory accesses at different strides

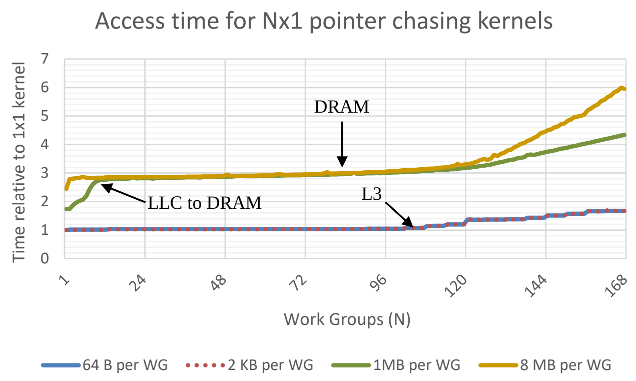


Fig. 10: Relative access time for Nx1 kernels chasing pointers in different working sets on SKL GPU

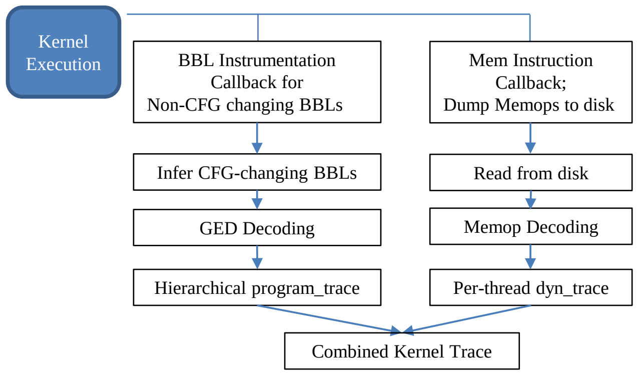

Kernel Execution
BBL Instrumentation Callback for Non-CFG changing BBLs
Mem Instruction Callback; Dump Memops to disk
Infer CFG-changing BBLs
Read from disk
GED Decoding
Memop Decoding
Hierarchical program_trace
Per-thread dyn_trace
Combined Kernel Trace

Fig. 11: GPU Trace Generation Overview

be expressed top down as Program →Thread List→ Kernel Invocation List→Function Invocation List→Basic Block List→ Instruction List. The hierarchical structure can be efficiently represented as hashmaps in the protobuf schema. The memory accesses are stored in a flat structure independent of the hierarchy since a single instruction may have different memory addresses at different times in the program. While reading the traces, the memory addresses are combined with the rest of the trace structure. We show a sample instruction as it would appear during the trace reading stage in Fig. 12. The instruction contains various fields decoded by GED. Some of the fields such as registers and memory addresses can be lists. The GEN ISA has a 2D addressing scheme for registers and memory. We flatten these out and express them as lists in the traces to make it easier to use the traces in simulators. The trace size is generally dependent on the amount of memory traffic in the application since the rest of the information is highly compressed due to the hierarchical nature. For the microbenchmarks described in the paper, the average trace size was under 2 bytes per instruction. We will release all the traces that we have collected using the tool publicly. The tool itself will also be published as open source once the underlying GTPin toolkit is available publicly.

V. Evaluation

We use the MacSim [5] heterogeneous architecture simulator for simulations, and add a new iGPU module to it. Currently, MacSim supports the x86 and ARM64 ISAs for CPU cores, and NVIDIA's intermediate PTX ISA for GPU cores. We extend MacSim to support Intel GPUs by integrating iGPU's traces, and creating an architectural model that is a hybrid of MacSim's internal CPU and GPU models.

```
opcode:                         GED_OPCODE_send
exec_size:                      16
dst_reg_file:                   GED_REG_FILE_GRF
dst_data_type:                  GED_DATA_TYPE_w
src0_reg_file:                  GED_REG_FILE_GRF
src0_data_type:                 GED_DATA_TYPE_uq
src1_reg_file:                  GED_REG_FILE_INVALID
src1_data_type:                 GED_DATA_TYPE_INVALID
dst_chan_en:                    16
dst_addr_imm:                   -1
ins_uid:                        "4_3_34"
src0_reglist:                   "39_0", "39_1", "39_2", ...
...                             ...
...                             ...
mem_addr:                       FF00007F04, FFB886CC88, ...
```

Fig. 12: Sample Instruction in the GPU Trace

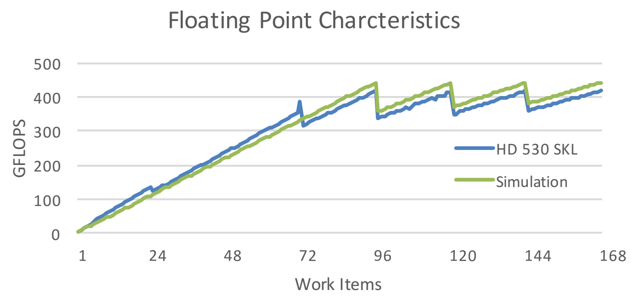


Fig. 13: Floating point SP simulation

Figure 13 represents the single precision floating point performance of the SKL GPU. As explained in Section III-A, it scales linearly in the beginning as we increase the work groups, but exhibits periodic dips in throughput due to the workload imbalance across EUs. We model such characteristics (the thread scheduler and compute units), and achieve very similar performance as the real GPU.

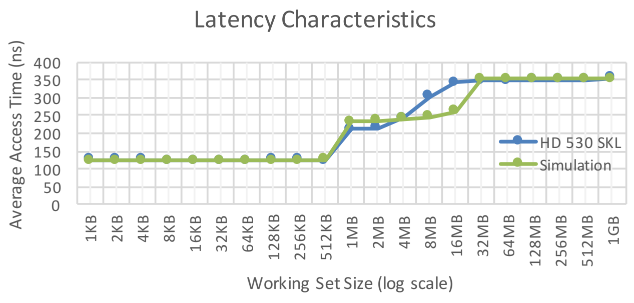


Fig. 14: Cache size simulation

Figure 14 represents the average access time at different working set sizes for the SKL GPU. We faithfully model the L3 and DRAM access latencies. The only deviation from the measured data is in the LLC region. As described earlier in Section III-B, the GPU is unable to use the full capacity, and shows a steady increase of access times in this region. This is not yet modelled in the simulator, and is a part of our future work.

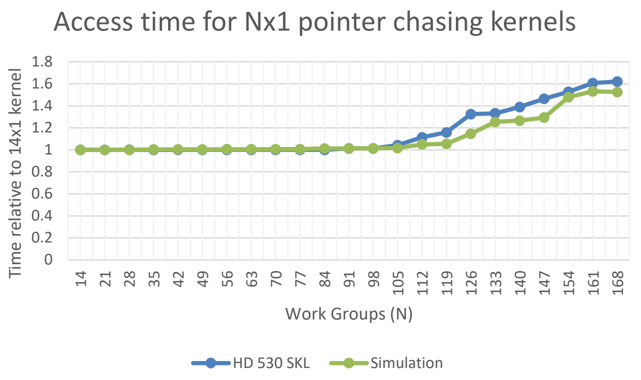


Fig. 15: MLP effect simulation. The working set is 2 KB per work group from Fig. 10

Figure 15 represents the degree of MLP for the SKL GPU. We are able to achieve similar trends of MLP close to 100. We model this behaviour in the simulator with structures such as the MSHR for caches. We show times relative to the 14x1 case as we are aware of a known bug in the simulator for very low thread counts.

VI. Related Work

Discrete GPUs have been studied extensively over the last decade due to a sustained interest in HPC and machine learning. H Wong et al. [11] study the microarchitecture of NVIDIA GeForce 200 GPUs through a series of CUDA microbenchmarks. Mei et al. [12] benchmark the memory hierarchy of NVIDIA's Fermi and Tesla GPUs across a variety of metrics. Taylor et al. [13] present a microbenchmark suite for AMD GPUs. More recently, commercial heterogeneous systems have spurred the interest of the community in understanding the memory access behaviour of these systems [14]. Daga et al. [15] study the efficacy of fused CPU-GPU architecture in addressing the PCI-E bottleneck. Resource sharing in such systems, particularly in the context of a shared LLC was studied by García et al. [16]. They find that applications modified to leverage the shared virtual space and fine grained synchronisations can lead to significant improvements. Hetero-Mark [17] is a new benchmark suite aimed specifically at collaborative computing in heterogeneous systems, and an exphasis on different memory patterns between CPUs and GPUs. On the simulation front, GPGPU-Sim [18] remains a popular simulator in the area of academic research for NVIDIA GPUs. It has also been integrated with the gem5 [19] CPU simulator to form a combined execution driven heterogeneous simulator, gem5-gpu [20]. A different approach to NVIDIA GPU simulation relies on the versatile GPUOcelot [21] dynamic compilation framework, which, among other things, facilitates PTX emulation. MacSim [5] simulator leverages this ability from GPUOcelot to simulate heterogeneous workloads. Among simulators that are not primarily focused on NVIDIA

GPUs, ATTILA [22] presents a general execution driven microarchitectural simulator for GPUs, and they simulate a subset of OpenGL calls. Multi2Sim [23] simulates the AMD Evergreen GPU's native ISA along with x86 GPUs in a heterogeneous simulator. Recently, some support for simulating AMD APUs with HSA workloads has also been added to gem5 [24]. Finally, for Intel GPUs, Kambadur et al. [3] present the foundational work on the GTPin toolkit and its use in instrumenting OpenCL applications. Their work focuses on traces of OpenCL calls to find representative kernels using SimPoint at kernel cycles granularity. Our work generalises the use of GTPin for generating standardised instruction level traces in an open source tool, which can be easily integrated into different cycle level architecture simulators.

VII. Conclusions

In this paper, we explored the microarchitectural characteristics of Intel Gen9 integrated iGPUs. We summarise the important findings about Intel GEN9 iGPUs from our experiments:

- 4 threads per EU are sufficient to reach peak GFLOPs.
- The GPU's memory hierarchy latency is higher than the CPU's, even when accessing the same resources like the LLC.
- The GPU is unable to use the LLC's total capacity.
- The GPU's accesses to the LLC are relatively unaffected by CPU's interference whereas the CPU's LLC accesses are affected by streaming interference from the GPU.
- The GPU has an MLP of around 100 memory requests across the memory hierarchy.

To the best of our knowledge, this is the first work that takes a detailed look at the Intel iGPU architecture, and provides a complete flow of modelling, trace collection, and simulation.

VIII. Acknowledgement

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported in part by Intel Corporation and the National Science Foundation under grant CCF-1054830.

References

[1] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 163–174.

[2] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan, "A framework for dynamically instrumenting gpu compute applications within gpu ocelot," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2011, p. 9.

[3] M. Kambadur, S. Hong, J. Cabral, H. Patil, C.-K. Luk, S. Sajid, and M. A. Kim, "Fast computational gpu design with gt-pin," in *Workload Characterization (IISWC), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 76–86.

[4] K. Varda, "Protocol buffers: Googles data interchange format," *Google Open Source Blog, Available at least as early as Jul*, vol. 72, 2008.

[5] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho, "Macsim: A cpu-gpu heterogeneous simulation framework user guide," *Georgia Institute of Technology*, 2012.

[6] S. Junkins, "The compute architecture of intel® processor graphics gen9," *Intel whitepaper v1*, 2015.

[7] "Micro48-tutorial on intel processor graphics: Architecture and programming — intel software," https://software.intel.com/en-us/blogs/2015/08/27/micro48-tutorial-on-intel-processor-graphics-architecture-and-programming, (Accessed on 07/13/2017).

[8] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "Pin: a binary instrumentation tool for computer architecture research and education," in *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*. ACM, 2004, p. 22.

[9] "Khronos releases opencl 2.2 with spir-v 1.2 - khronos group press release," https://www.khronos.org/news/press/khronos-releases-opencl-2.2-with-spir-v-1.2, (Accessed on 05/30/2017).

[10] M. Stephenson, S. K. Sastry Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler, "Flexible software profiling of gpu architectures," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 185–197.

[11] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 235–246.

[12] X. Mei, K. Zhao, C. Liu, and X. Chu, "Benchmarking the memory hierarchy of modern gpus," in *IFIP International Conference on Network and Parallel Computing*. Springer, 2014, pp. 144–156.

[13] R. Taylor and X. Li, "A micro-benchmark suite for amd gpus," in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*. IEEE, 2010, pp. 387–396.

[14] J. Hestness, S. W. Keckler, and D. A. Wood, "A comparative analysis of microarchitecture effects on cpu and gpu memory system behavior," in *Workload Characterization (IISWC), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 150–160.

[15] M. Daga, A. M. Aji, and W.-c. Feng, "On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing," in *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*. IEEE, 2011, pp. 141–149.

[16] V. García, J. Gomez-Luna, T. Grass, A. Rico, E. Ayguade, and A. J. Pena, "Evaluating the effect of last-level cache sharing on integrated gpu-cpu systems with heterogeneous applications," in *Workload Characterization (IISWC), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 1–10.

[17] 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 *Workload Characterization (IISWC), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 1–10.

[18] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, april 2009, pp. 163 –174.

[19] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[20] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous cpu-gpu simulator," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 34–36, 2015.

[21] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan, "A framework for dynamically instrumenting gpu compute applications within gpu ocelot," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: ACM, 2011, pp. 9:1–9:9. [Online]. Available: http://doi.acm.org/10.1145/1964179.1964192

[22] V. M. Del Barrio, C. González, J. Roca, A. Fernández, and E. Espasa, "Attila: a cycle-level execution-driven simulator for modern gpu architectures," in *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*. IEEE, 2006, pp. 231–241.

[23] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: a simulation framework for cpu-gpu computing," in *Parallel Architectures and Compilation Techniques (PACT), 2012 21st International Conference on*. IEEE, 2012, pp. 335–344.

[24] "Microsoft powerpoint - 03 amd-apu-model.pptx," http://www.gem5.org/wiki/images/7/7a/2015_ws_03_amd-apu-model.pdf, (Accessed on 10/17/2017).