Proceedings of 24th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), Heraklion, Greece, 2016

Microbenchmarks for GPU characteristics: the occupancy roofline and the pipeline model.

Jan Lemeire[1,2,3], Jan G. Cornelis[2,3], Laurent Segers[1]
[1]Vrije Universiteit Brussel (VUB), Industrial Sciences (INDI) Dept.,
Pleinlaan 2, B-1050 Brussels, Belgium
jan.lemeire@vub.ac.be
[2]Vrije Universiteit Brussel (VUB), Electronics and Informatics (ETRO) Dept.,
Pleinlaan 2, B-1050 Brussels, Belgium
[3]iMinds, Multimedia Technologies Dept.
Gaston Crommenlaan 8 (box 102), B-9050 Ghent, Belgium

***Abstract*—In this paper we present microbenchmarks in OpenCL to measure the most important performance characteristics of GPUs. Microbenchmarks try to measure individual characteristics that influence the performance. First, performance, in operations or bytes per second, is measured with respect to the occupancy and as such provides an *occupancy roofline* curve. The curve shows at which occupancy level peak performance is reached. Second, when considering the cycles per instruction of each compute unit, we measure the two most important characteristics of an instruction: its issue and completion latency. This is based on modeling each compute unit as a pipeline for computations and a pipeline for the memory access. We also measure some specific characteristics: the influence of independent instructions within a kernel and thread divergence. We argue that these are the most important characteristics for understanding the performance and predicting performance. The results for several Nvidia and AMD GPUs are provided. A free java application containing the microbenchmarks is available on www.gpuperformance.org.**

***Index Terms*—microbenchmarks, GPU, opencl, performance**

I. Introduction

The goal of this paper is to extract the important GPU performance parameters with microbenchmarks. Microbenchmarks are small programs especially designed to measure a single parameter, in contrast with benchmarks that estimate the performance for real applications.

The roofline model [9] is a simple approach for the performance prediction of a code run on a parallel platform. It is based on the computational and communication peak performance to determine the number of computation and communication cycles. It assumes that the execution of both subsystems is perfectly overlapped and predicts the actual runtime to be the maximum of both. To determine whether a code is computation or communication bound, the arithmetic intensity (the ratio of computation to communication) is used. The roofline model uses *ceilings* to account for inefficient usage of the GPU. These ceilings, however, are software independent and cannot account for all sources of overhead. The roofline model has been adapted for GPUs. The best known examples are the boat hull model of [5] and GPURoofline of [3] and, more recently, the quadrant-split model of [4]. These approaches comply with our first set of microbenchmarks. However, peak performance and perfect overlap of computation and communication cannot be guaranteed; a second set of microbenchmarks is necessary.

Although GPUs offer a performance that is 50 to even 200 times greater than CPU performance, this is restricted to fine-grain data parallel code. This is code with a lot of independent work where every unit of work (called work item) takes place on local data and uses similar operations as the other units and where furthermore few resources are needed to process one work unit. A major performance parameter is the number of threads that can run simultaneously (concurrently) on a Compute Unit (CU). The CU, called streaming multiprocessor on Nvidia GPUs, is the basic computing component on a GPU, similar to a core on a CPU. This is called the *occupancy* of a GPU. A higher occupancy ensures that all processing elements are busy during each clock cycle through latency hiding. This is shown in Fig. 1 in which each work item is executing the same number of instructions. We gradually increase the number of work items. One observes that the execution time remains constant up to 2000 work items, since they will run concurrently. Only when all processing pipelines are filled, the execution time starts to increase.

The first set of benchmarks that we built are providing the *occupancy roofline* for each computational and memory instruction type. It shows the increase of performance with the increase of concurrency and the convergence point at which the peak performance is reached.

Another goal of this paper is building microbenchmarks for retrieving the GPU characteristics that determine the performance of GPU programs in absence of full occupancy. The program is then called latency-bound, since the latencies of the GPU will determine the performance. Performance models for such cases were designed by [1] and [2]. The goal of these models is not to perform cycle-accurate simulations but to attain a good balance between simplicity, generality and accuracy. We argue that the main parameters for such models are the issue and completion latencies of instructions and memory requests. Consider a basic pipelined processor. The issue latency is the time required between issuing two independent instructions. This is one cycle for a simple

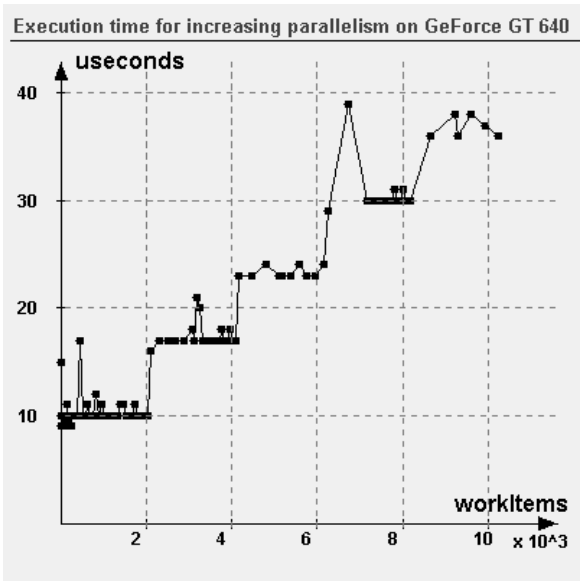

Fig. 1. Execution time for the same kernel run with different number of work items (NVidia Geforce).

pipeline. The completion latency is the time it takes for an instruction to complete, i.e. the length of the pipeline. We will motivate that more complex pipelines as in modern GPUs can also be adequately characterized with both latencies, if one accepts that the latencies cannot be mapped directly on single hardware parameters. Although sometimes implicit, both approaches for performance prediction [1], [2] are based on the issue and completion latencies. In the work of [2] the issue latency is called departure delay. They also designed a microbenchmark to measure the departure delay, but it partly relies on knowledge of some other hardware parameters. Our approach on the other hand wants to take as few assumptions as possible about the underlying hardware. Another related work are the microbenchmarks of [4], but they focus entirely on kernels that are not latency-bound. Also [10] uses microbenchmarks to understand the performance characteristics of GPUs. They estimate peak performance and the necessary concurrency to fully utilize the pipeline. But their main focus is on special characteristics such as the effect of synchronization and serialization.

First we discuss the GPU architecture, then we introduce the occupancy roofline. In section 4 we define and discuss the latencies of the pipeline model. Based on the definitions, we present microbenchmarks to measure the latencies for computational instructions (Section 5) and for memory requests (Section 6). This results in a detailed report of the GPU performance characteristics (Section 7).

II. The GPU architecture

We first introduce a general architecture of most modern GPUs, on which the performance analysis of this paper is based. The level of detail is chosen such that it can model a broad class of GPUs without getting lost in vendor or architecture specific details. The model is borrowed from OpenCL and can easily be mapped on a typical GPU.

TABLE I
Hardware and software parameters of a GPU.

Hardware parameters	Software parameters
$ClockFrequency$	$workItems$
$NbrPE$	$WGSize$
$NbrCU$	$localMem$
$MaxLocalMem$	$instrKernel$
$MaxConcWG$	$memInstrKernel$
$WarpSize$	$bytesPerElement$

A GPU is connected to an external RAM memory that we refer to as the global memory. It typically has a few GBs. A GPU consists of a number of identical Compute Units (CUs), denoted by $NbrCU$. On Nvidia GPUs they are called streaming multiprocessors. Each CU has its own *local memory* with size $MaxLocalMem$, which can be regarded as cache memory. Code to be executed on a GPU is specified by writing a kernel that describes what one *work item* must do. The programmer will choose a sufficient number of work items, $workItems$, to carry out all the work. Each work item will execute the code specified in the kernel and use identifier functions to find out what data to work on. The instructions of the kernel will be performed by the processing elements of the CU. The number of processing elements per CU for the instruction under study is denoted by $NbrPE_{itype}$ where $itype$ denotes the instruction type.

Work items are organized in work groups. The size of a work group (WG), $WGSize$, is chosen by the user. A work group is executed on a single CU. Depending on the register and local memory usage ($localMem$) of a work group and as long as the maximum number ($MaxConcWG$) is not reached, several work groups can run concurrently on the same CU. We call this parameter $concWG$. Work items within the same work group are executed in warps (Nvidia) or wavefronts (AMD) in lockstep (SIMT). We will use the term *warp* throughout the paper. The warp size is a fixed hardware parameter ($WarpSize$). The CU scheduler considers the pool of warps (from all concurrent WGs) and selects a warp that has an instruction for which all dependent instructions have completed (modern GPUs will schedule independent instructions within the same kernel).

Table I summarizes the hardware and software parameters used throughout the paper. Hardware parameters start with a capital.

We denote the number of computational and memory instructions under study (see later) by $instrKernel$ and $memInstrKernel$ respectively. The element size in Bytes of a memory instruction is denoted by $bytesPerElement$.

III. Computational Peak Performance: Occupancy roofline

Here we study the computational peak performance for each instruction type. Given that each processing element

can execute 1 instruction every cycle, the theoretical peak performance is:

$$PeakPerf_{itype} = NbrCU.NbrPE_{itype}.ClockFrequency\,. \quad (1)$$

For most GPUs, this value can be multiplied by 2 for floating-point multiply-add instructions since they are executed in the same cycle. Furthermore we study the performance in function of the occupancy, which results in a curve having the form of a roof. The *occupancy roofline* is defined as the computational performance in function of the number of concurrent work items on one CU. The latter defines the occupancy.

The microbenchmark for measuring the occupancy roofline executes many instructions of the selected type in an unrolled for-loop. The number of iterations is chosen large enough to make the impact of overheads very small. To avoid compiler optimizations we make the instructions data-dependent, the loop starts with a value of an input array. To make sure that the compiler will not eliminate useless instructions, we pretend that the result is written back to an output array. We add a condition based on a boolean kernel parameter (which is set to false at runtime) and the computed value so that no actual write back is performed. The number of concurrent work items is set by choosing the work group size $WGSize$, controlling the number of concurrent work groups by choosing the amount of local memory each work group needs.

$$localMem = \lfloor MaxLocalMem/concWG \rfloor \quad (2)$$

The same experiment is performed 25 times. The average and the 95% confidence interval is reported. The latter is calculated as 1.96 times the standard deviation of the experimental values.

We do this for the different types of instructions:

- floating point instructions (SP),
- floating point multiply-add instructions (MADD),
- integer instructions (INT),
- double-precision (DP) (when supported by the GPU),
- native Special Functions (SF) instructions (e.g. $native\_cos$),
- SF instructions without hardware support, which we call software-sf (e.g. cos).

Fig. 2 shows the occupancy roofline model for the different instruction types for a GeForce GT 640. The theoretical peak performance (Eq. 1) is 361.2Gops since it has 2 CUs, 192 PEs per CU (Fermi architecture) and a clock frequency of 900MHz. For single-precision multiply-additions, 691 Gflops can be attained theoretically. This performance is not attained in this experiment since each instruction in the kernel depends on the result of the previous; there are no independent instructions that can be issued simultaneously. With 2 and 4 independent instructions per iteration a peak performance of respectively 611 and 624 Gflops is measured.

From the roofline graph we can identify how much concurrency is needed to fill the pipelines (and hence maximize latency hiding). The curve converges to the peak performance

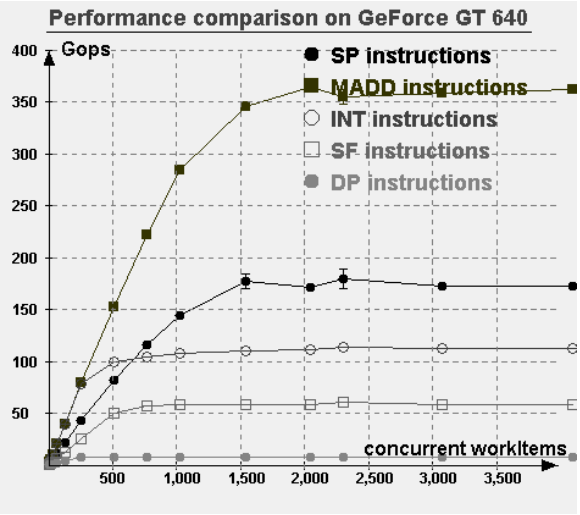

Fig. 2. Performance for different instruction types.

at the *ridge point*. In practice, we take the minimal point (in number of work items) at which the relative difference with the peak performance is smaller than 5%. Section 7 shows the empirical numbers in a report.

IV. Microbenchmarks for latencies based on the pipeline model

The second set of microbenchmarks focusses on measuring the relevant latencies of a GPU. We propose to use an abstract pipeline model that is characterized by 2 latencies: issue and completion latency. The pipeline model is discussed here. The benchmarks are discussed in the following sections.

A. The pipeline model

The model is based on an abstract pipeline for modeling each subsystem of a Compute Unit (CU): one for the computational subsystem and one for the memory subsystem. A pipeline is mainly characterized by two parameters: the number of cycles required between issuing two consecutive independent instructions (called *issue latency*, denoted by λ) and the number of cycles until the result of an instruction is available for use by a subsequent instruction (called operation latency or *completion latency*, denoted by Λ) [8]. Both latencies are in most cases sufficient to estimate the execution time of a program: the issue latency is the time before the next independent instruction can be started, the completion latency is the time before the next dependent instruction can be started if all other instructions it depends on have completed as well.

Peak performance is attained with full pipelines, which relies on hiding the completion latencies. Latency hiding is achieved by executing independent instructions (from the same thread or from a different thread) at the same time: the completion latencies will be (partially or completely) hidden. When completely hidden, the issue latencies determine the run time and we attain peak performance.

For a scalar pipeline the issue latency is one clock cycle: after one clock cycle the instruction enters the next stage and a new instruction can enter the pipeline. The completion latency is the number of stages in the pipeline multiplied by the issue latency. This is no longer true for modern processors which are much more complex.

B. Complex GPUs

Each instruction or memory transaction is characterized by its issue and completion latency, also when more complex processors are considered. We argue that it is appropriate to model complex GPUs with the same 2 pipelines, but use more complex latencies that may even depend on the context. The latencies should not be integers. They can also depend on the context of the instruction in which case they are not constants.

Let's discuss some more detailed characteristics of modern GPUs and show how they can be modeled by both latencies.

Within a CU, different instruction types can have different Processing Elements (PEs) at their disposal. This is modeled by considering different latencies for each instruction type. Also for memory instructions we will use different latencies for different memory levels. The effect of caching can be considered as context (whether the data is cached or not) that determines the completion latency.

So-called superscalar pipelines can execute more than one instruction at the same time. On a GPU a warp of work items is executing the same kernel in lockstep. Only one instruction is scheduled which is executed for all $WarpSize$ work items. This is called Single Instruction Multiple Thread (SIMT) where the execution of a work item can be considered a thread. The pipeline model is about scheduling computational or memory instructions. Therefore we have to consider a warp as the fundamental scheduling unit or 'thread'. In the remainder of the text we will use the term 'warp' to denote the basic scheduling unit (and not the term 'thread' because it may result in confusion). It is therefore more interesting to consider the latencies for a warp and not for a work item. An Nvidia GPU of the Tesla generation has 8 floating point PEs, giving an issue latency of 1/8 for a floating point instruction. If we consider a warp, the issue latency is 4: each 4 cycles, a warp can be scheduled.

Each CU of an Nvidia GPU of the Kepler generation has 4 warp schedulers each having 2 instruction dispatchers, which makes that 2 independent instructions of 4 ready warps can be scheduled. This results in a warp issue latency of 1/8. Note that this is only achieved with a good mix of instructions since the number of processing elements per instruction type is limited. A Kepler CU has 192 single precision processing elements, so that maximally 6 floating point instructions can be scheduled together (warp issue latency of 1/6).

Lastly, overheads due to inefficient execution of instructions may induce larger latencies. For SIMT it is not always possible to execute all instructions of a warp simultaneously. In some cases instructions or memory requests must be serialized. With bank conflicts for instance, the memory cannot serve the concurrent memory requests instantly. Requests that cannot be served together have to be served one after the other (serialized) because local memory is partitioned into banks. Each bank can only handle one memory request at the same time.

C. Performance understanding and prediction

We believe that the resource usage is effectively modelled with the abstract pipeline, in the sense that it leads to a good balance between accuracy and simplicity. The model can be used to understand, predict or analyze the performance. The effect of each instruction *individually* is modelled by both latencies, while the *interplay* of the different instructions is modelled by the pipelines: an independent instruction can be scheduled after the issue latency of the previous instruction. A dependent instruction can be scheduled when all instructions it depends on have completed. The exact moment is determined by the completion latencies of said instructions.

V. Computational instruction latencies

In this section we discuss how we measure both lambdas for computational instructions.

A. Microbenchmarks

Measuring the completion latency for computational instructions is straightforward: one runs a single thread in which all instructions are dependent. To achieve this we make a work group of 1 work item and ensure that only 1 work group will be run simultaneously on a multiprocessor. Therefore we allocate a block of local memory that occupies almost completely the local memory of a multiprocessor, as expressed by Eq. 2. Then the number of Cycles Per Instruction (CPI) reflects the completion latency.

Measuring the issue latency is more difficult: we have to ensure that the pipeline is completely full. In attempting to reach full latency hiding, one has to exploit two levels of parallelism: Warp-Level Parallelism (WLP) and Instruction Level Parallelism (ILP), which is achieved by independent instructions within a single kernel. We increment the work group size (a runtime parameter) and the number of concurrent work groups by lowering the local memory needed by a single work group. We employ kernels with a different number of independent instructions.

The first curve (full circles) of Fig. 3 shows the result for a kernel with all dependent integer instructions: the CPI in function of the number of concurrent work items. The leftmost point reflects the completion latency (Λ) or, in the case of a simple pipeline, the number of pipeline stages the instruction needs to traverse until it is completed and the scheduler is ready to issue another dependent instruction. For the Nvidia GeForce GT 640 it is 6 cycles. By increasing the concurrency, the number of cycles per instruction decreases due to the filling of the pipelines and the latency hiding. It converges to 0.5 cycles (λ). Kepler architectures have 32 PEs per CU for integers and can execute an integer multiply-add as a single instruction. Our benchmark was an iteration of multiplications and additions.

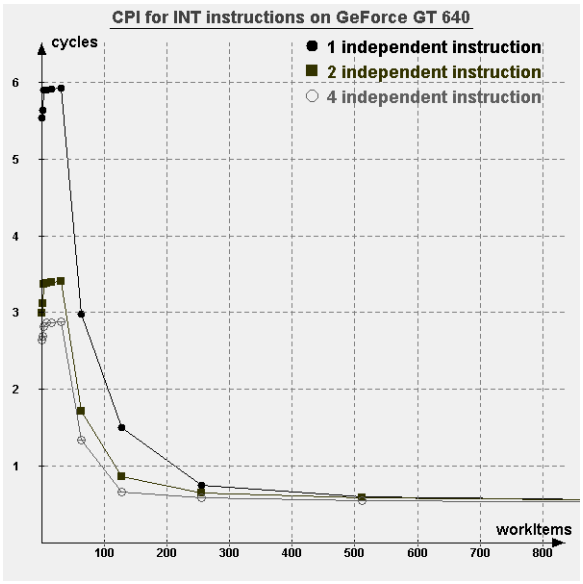

Fig. 3. CPI(warp) in function of concurrent workitems for 1, 2 or 4 independent integer instructions in kernel (Nvidia GeForce GT640).

Note that this graph is a kind of upside down occupancy roofline.

B. Equations

We derive equations to calculate the CPI from the measured runtime of a microbenchmark. We start by an approximated value. The computational performance, expressed in operation per second, is given by the following equation:

$$\begin{aligned} performance &= nbrInstructions/runtime \\ &= instrKernel * workItems/runtime \end{aligned}$$

Given the processor's clock frequency, the number of Cycles Per Instruction (CPI) can be calculated:

$$CPI_{GPU} = runtime * ClockFrequency/nbrInstructions$$

The approximated CPI of a single Compute Unit (CU) is then:

$$CPI'_{CU} = CPI_{GPU} * NbrCU \tag{3}$$

CPI'_{CU} is more useful than CPI_{GPU} because GPUs of the same generation use the same CUs, which makes the results portable across all GPUs of the same generation. It will also be easier to map the latencies on the hardware parameters of the CU.

These equations are only correct if the GPU is fully utilized during the whole execution. This is not always the case as shown in Fig. 4. We call the concurrent execution of a set of work groups by a compute unit a *run* (gray box in figure). As such, the execution can be regarded as the distribution of the runs over the available compute units. Here we assume that all concurrent work groups start and stop at the same time. At the end of the execution, there may be less runs left than compute units or a run may not be occupied by the same number of work groups. This will result in a lower performance. To correct for this, we base the calculation of the performance variables on the execution time of one run, denoted by $timeOfRun$. This leads to the following equations:

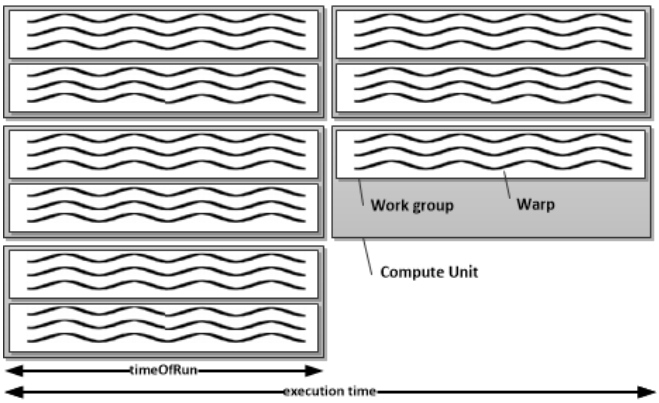

Fig. 4. Execution of the work groups of warps by the compute units of a GPU. Here we have 3 compute units that each run 2 work groups concurrently. Each work group consists of 3 warps.

$$WG = \lceil workItems/workGroupSize \rceil$$
$$concWG = \min(MaxConcWG, WG, \frac{MaxLocalMem}{localMemSize})$$
$$warpsWG = \lceil WGSize/WarpSize \rceil$$
$$actualWarpSize = WGSize/warpsWG$$
$$concWarps = \min(MaxConcWarps, warpsWG * concWG)$$
$$totalWarps = workItems/actualWarpSize$$
$$runsCU = \lceil totalwarps/NbrCUs/concWarps \rceil$$
$$timeOfRun = runtime/runsCU$$
$$cyclesOfRun = timeOfRun * ClockFrequency$$
$$instrRun = instrKernel * workGroupSize * concWG$$
$$CPI_{CU} = cyclesOfRun/instrRun$$
$$CPI_{warp} = CPI_{CU} * actualWarpSize$$

Based on these definitions, we can empirically extract the latencies for an instruction of type $itype$ as

$$\lambda^{itype} = \min_{exp} CPI^{itype}_{warp} \tag{4}$$
$$\Lambda^{itype} = \max_{exp} CPI^{itype}_{warp} \tag{5}$$

where the minimum and maximum is taken over a set of experiments varying from 1 concurrent warp to full occupancy.

Table II shows the obtained latencies for different generations of GPUs.

C. Instruction-Level Parallelism

The presence of independent instructions in a kernel makes it possible for the instruction scheduler to issue more than one instruction of the same warp without having to wait for their completion. Fig. 3 shows the CPI in function of the occupancy

TABLE II
ISSUE AND COMPLETION LATENCIES FOR SINGLE POINT FLOATS (SP), DOUBLE PRECISION (DP) AND SPECIAL FUNCTIONS (SF) FOR THE THREE GENERATIONS OF NVIDIA GPUS AND TWO TYPES OF AMD GPUS (SOUTHERN ISLANDS AND NORTHERN ISLANDS).

	Kepler		**Fermi**		**Tesla**		**Southern**		**Northern**	
	Λ	λ	Λ	λ	Λ	λ	Λ	λ	Λ	λ
SP	9	0.18	18	1	22	4	6	1	8	2.7
DP	32	4	22	2	-	-	22	5	32	8
SF	18	1	40	8	51	16	29	7	51	10

for 3 kernels with respectively 1, 2 and 4 independent integer instructions. The body of the loop of the benchmarks consists of respectively 1, 2, 4 instructions that only depend on the previous iteration's results.

In the case of a single concurrent warp (first point at the left) we observe that a lower CPI is attained, showing that the GPU architecture is able to exploit the independencies by filling the pipelines with independent instructions. On the other hand, the CPI of the 3 kernels converges to 1/2 cycle, which means that the presence of independent instructions is not necessary to fully occupy the pipelines because the pipeline needs less concurrent warps to be filled.

D. Thread divergence

Most GPUs execute work items in warps, in which each member executes the same instruction. In case of conditional instructions (if or while statements), instructions are issued for the whole warp but only executed for those work items for which the condition holds. So with an if-then statement, the instructions after the if-condition *and* after the then are issued but only executed for some work items. Due to code divergence within a warp, some performance is lost. If for instance the if and then part have the same number of instructions, then performance is halved for that part of the code. If all work items of a warp follow the same branch, then the instructions of the unfollowed branches are not issued.

To test this effect we created a benchmark that consists of 4 branches each having the same number of computational instructions (enough to hide all overhead). Each work item follows one branch according to its global id. We vary the number of consecutive work items that follow the same branch, denoted by parameter $convItems$. The branch of a work item is chosen as $(get\_local\_id(0)/convItems)\%4$. The performance in function of $convItems$ for an Nvidia GPU is shown in Fig. 5. Performance reaches its maximal value at a value of 32, since then all work items of warp 1 are following the first branch, warp 2 the second branch, and so on. Absence of divergence within each warp happens for multiples of the warp size.

VI. MEMORY REQUEST LATENCIES

When estimating the latencies of memory requests, we have to consider different memory types (global, local, constant and private memory) and different element sizes (1, 2, 4, 8, 16 bytes). Furthermore, each memory type can have a cache.

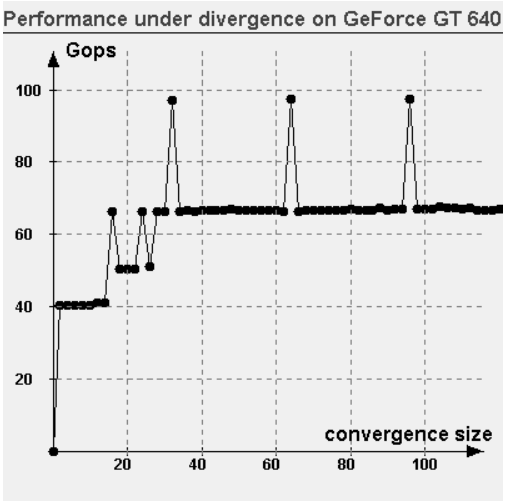

Fig. 5. Performance of kernels with thread divergence.

Each characteristic is measured with a specific microbenchmark:

- Bandwidth and Issue latency: *Copy-kernel*
- Completion latency: *Random walk*
- Cache behavior: *Varying-size random walk*

A. The copy-kernel

To measure the issue latency, we copy data from an input array to an output array. The impact of overheads, such as the computational instructions needed for calculating the address, is minimized by letting each work item read several elements (in an unrolled for-loop). To ensure that maximal bandwidth is attained, the data should be read in a regular way that complies with the hardware. We let work items read consecutive elements: work item 1 reads element 1, work item 2 reads element 2, etcetera. In the next iteration, each work items adds the number of work items to the address it is going to read. This is shown in the code below.

```
#define N 100
// src: array with N * NBR_WORK_ITEMS elements
// dst: array with NBR_WORK_ITEMS elements
__kernel void copy(__global float *src,
    __global float *dst, int flag)
{
    int id = get_global_id(0);
    int index = id;
    int NBR_WORK_ITEMS = get_global_size(0);
    float sum = 0;

    #pragma unroll
    for (int i = 0; i < N; i++) {
        sum += src[index];
        index += NBR_WORK_ITEMS;
    }

    if (flag * sum)
        dst[id] = sum;
}
```

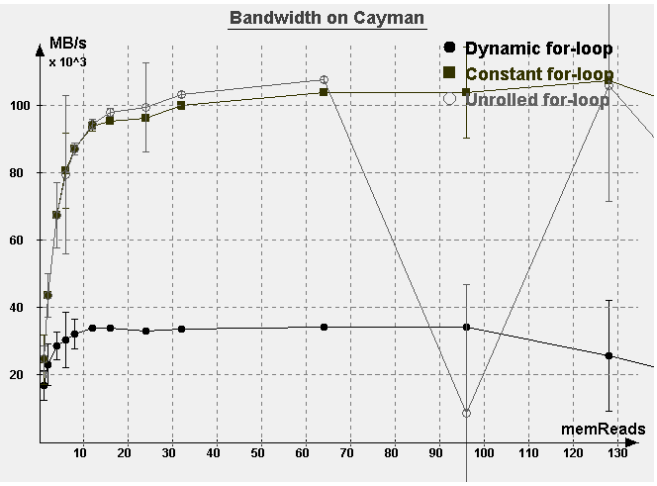

Fig. 6. Bandwidth for different memory reads per kernel (AMD Radeon 6900 Cayman GPU).

GPUs can be regarded as massive 'thread' processors for which threads do not cause overheads (this in contrast with the large context switch overhead on CPUs). Having a lot of work items with each just a small amount of instructions is in theory as efficient as few work items with each having more work. To verify this assumption, we vary the number of read instructions of the copy-kernel. The results are shown in Fig. 6. We compare a kernel based on a loop determined by a kernel parameter ('dynamic loop') with that of a constant (defined at compile time) determining the number of iterations and an unrolled for-loop (shown in the listing above). The results show that a parameterized loop ('dynamic loop') attains a much lower bandwidth then a constant loop. Explicit loop unrolling, however, does not offer a substantial additional performance increase. The graph also shows some noisy periods in which the performance fluctuates a lot. This was especially true for the AMD Cayman GPU.

B. The random walk

To accurately measure the completion latency we have to ensure that two consecutive memory reads in a kernel are dependent. This is accomplished by a *chasing pointer* algorithm applied on a given array with $arraySize$ elements that resides in a particular memory type. The array is traversed by using the element's value as the index of the next element:

```
int num_iterations = 1000;
int index = get_global_id(0) % arraySize;
while (num_iterations--)
        index = array[index];
```

We generate a random memory access pattern such that there can be no benefit of prefetching caches (as speculative loads become impossible) and cache-line hit rates reduce. We also have to ensure that the whole array is traversed. Therefore the array randomization is performed by swapping two random elements as shown in the following listing:

```
// initialisation step: encode index as values
for (int index = 0; index < length; ++index)
        array[index] = index;
// shuffle the array
int index, swap_pos, swap_val;
for (index = 0; idx < length - 1; ++index) {
        swap_pos = random(index, length);
        swap_val = array[index];
        array[index] = array[swap_pos];
        array[swap_pos] = swap_val;
}
```

The function $random(min, max)$ randomly generates a value in the range $[min, max[$. The algorithm is a minor variation on the Fisher-Yates shuffle known as Satollo's algorithm [7]. It generates an array that encodes a complete pass through the whole array. It is paramount for the pass to equal the length of the array, as some memory locations may be left unread otherwise. This would cause inaccurate results as the memory range subject to the test could be smaller than requested. In-depth studies on more intricate properties of Satollos algorithm are available in literature, e. g. Helmut Prodinger presents a detailed mathematical analysis on two parameters, 'the number of moves' and 'distance', in [6].

The completion latency of a memory access is measured similar to that of computational instructions: it is given by the number of Cycles Per Memory Request (CPMR) when 1 concurrent warp is run.

Besides measuring the latencies for global memory, we also apply it to local, constant and private memory. By private memory we mean arrays defined as local variables in a kernel. These arrays do not reside in registers as simple local variables do, but will be stored in global memory and may be cached. The benchmarks for local and private memory first copy the randomized array from global memory before starting the random walk. The choice of the array size is discussed in the next subsection, since we will measure the cache latencies at the same time.

C. Varying-size random walk

Cache sizes and latencies can be identified by using different array sizes during random walk experiments. As long as the array fits in the cache, the complete array will reside in the cache after 1 pass through the whole array. We perform multiple passes to measure the cache completion latency. By increasing the array size, the CPMR increases as a rounded step function, a step for each cache level. As soon as the array does not fit any longer in the cache, there will be cache misses, but still some cache hits as well, since by chance the data might be in the cache. The cache hit ratio is the cache size divided by the array size.

$$CPMR = \Lambda_{L1} + (\Lambda_{L2} - \Lambda_{L1})(1 - \frac{CacheSize}{arraySize}) \qquad (6)$$

This explains the rounded step function for the constant memory latencies shown in Fig. 7. The cache has a size of 8 KB. These experiments give us the completion latencies for the memory under study and show also the cache hierarchies, their sizes and latencies. It can be seen that arrays stored in private memory are not cached.

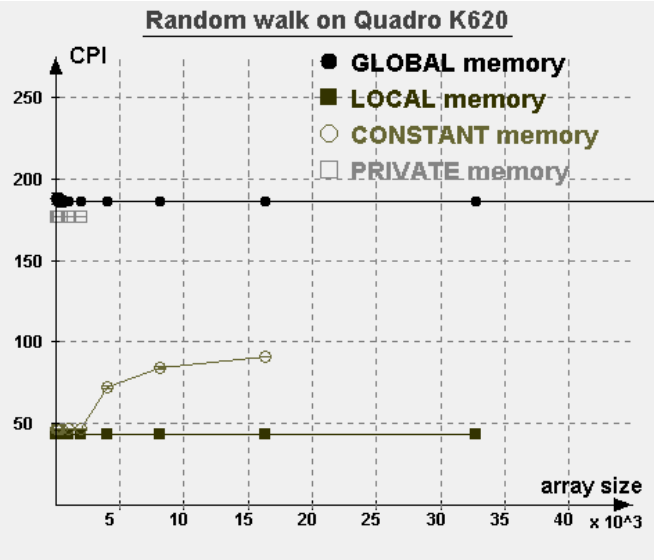

Fig. 7. Varying-size random walk on Nvidia Quadro K620.

TABLE III
PERFORMANCE REPORT OF A NVIDIA QUADRO K620 BASED ON THE MICROBENCHMARKS. PERFORMANCE IS IN GOP/S, BANDWIDTH IN GB/S, LATENCIES IN CYCLES.

Computations				
itype	Performance	λ	Λ	ridge point
SP	249/363/388	.43/.30/.28	9.1/4.7/4.7	1536
INT	125/130/130	.86/.77/.83	16.2/12.9/12.6	1536/1024/512
SF	27/27/27	4.0/4.0/3.9	62/39/29	512/384/256
Global Memory				
Element	Bandwidth	λ	Λ	cache size
1B	15.7	3.96	42.4	-
2B	21.6	5.83	59.7	-
4B	25.0	9.89	52.5	-
8B	26.2	18.9	78.5	-
16B	26.4	37.7	144.2	-
Local Memory				
	Bandwidth	λ	Λ	cache size
4B	138	1.3	43.1	-
Constant Memory				
	Bandwidth	λ	Λ	cache size
4B	97.1	1.48	48/90.6	8KB
Private Memory				
	Bandwidth	λ	Λ	cache size
4B	-	-	177	-

D. Equations

The bandwidth and CPMR can be calculated in a straightforward way:

$$
\begin{aligned}
bytesComm &= workItems * memInstrKernel * bytesPerElement \\
bandwidth &= bytesComm/runtime \\
totalMemInstr &= workItems * memInstrKernel \\
CPMR^{'} &= runtime * ClockFrequency/totalMemInstr
\end{aligned}
\tag{7}
$$

As is the case with the computational instructions, the CPMR can be calculated more precisely as follows:

$$
\begin{aligned}
memInstrRun &= memInstrKernel * WGSize * concWG \\
CPMR_{CU} &= cyclesOfRun/memInstrRun \\
CPMR_{warp} &= CPMR_{CU} * actualWarpSize
\end{aligned}
$$

VII. GPU PERFORMANCE CHARACTERISTICS REPORT

Based on the microbenchmarks we developed a benchmark suit that produces a report of a GPU's major performance characteristics. The report for the Nvidia Quadro K620 is shown in Table III. Multiple values for the computational latencies refer to the latencies for ILP levels of 1, 2 and 4. Multiple values for the memory latencies refer to the latencies of the different memory levels and indicate the presence of caches. The cache sizes are given in the last column.

VIII. CONCLUSIONS

One of the challenges of GPU computing is writing efficient programs. For this, knowledge of the architecture is indispensable. But as not all hardware details are disclosed by GPU vendors, we built microbenchmarks to determine them empirically. Our approach is to characterize computational instructions and memory requests by their issue and completion latency. These parameters can be used to understand and predict the performance based on performance models. A java application containing all benchmarks is freely available on www.gpuperformance.org.

This research has been funded by the Flemish 'Instituut voor Wetenschap en Technologie' (IWT) in the context of the ITEA-2 MACH project (IWT130177): 'Massive Calculations on Hybrid Systems'.

REFERENCES

[1] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen mei W. Hwu. An adaptive performance modeling tool for gpu architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 105–114, New York, NY, USA, 2010. ACM.

[2] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *ACM SIGARCH Computer Architecture News*, 37(3):152–163, 2009.

[3] Haipeng Jia, Yunquan Zhang, Guoping Long, Jianliang Xu, Shengen Yan, and Yan Li. Gpuroofline: A model for guiding performance optimizations on gpus. In *Euro-Par 2012 Parallel Processing - 18th International Conference*, pages 920–932, 2012.

[4] Elias Konstantinidis and Yiannis Cotronis. A practical performance model for compute and memory bound GPU kernels. In *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015*, pages 651–658, 2015.

[5] Cedric Nugteren and Henk Corporaal. The boat hull model: adapting the roofline model to enable performance prediction for parallel computing. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012*, 2012.

[6] Helmut Prodinger. On the analysis of an algorithm to generate a random cyclic permutation. *Ars Combinatoria*, 65:7578, 2002.

[7] Sandra Sattolo. An algorithm to generate a random cyclic permutation. *Information processing letters*, 22(6):315317, 1986.

[8] John Paul Shen and Mikko H. Lipasti. *Modern Processor Design*. McGraw-Hill, 2005.

[9] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[10] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2010*, pages 235–246, 2010.