A Cache-Aware Performance Prediction Framework for GPGPU Computations

Alexander Pöppl[✉] and Alexander Herz

Institut für Informatik, Technische Universität München,
Boltzmannstraße 3, 85748 Garching Bei München, Germany
{poeppl,herz}@in.tum.de

Abstract. We present a model for the automated prediction of average execution times for OpenCL-based computations on GPUs. The model encompasses the whole execution of the computation, including the transfer to and from the GPU, and the kernel execution itself. In contrast to existing static prediction frameworks, we incorporate the caches available on modern GPUs into our model. Using our benchmark suite, we show that memory access patterns can be grouped into five patterns that exhibit significantly different memory access performance. By extending our static analysis framework to differentiate the performance behavior of these memory access patterns, we improve on predictions made by existing GPU performance models. In order to evaluate the quality of our model, we compare cache-aware and cache-unaware predictions for a large set of randomly generated OpenCL kernels with their actual execution time.

1 Introduction

OpenCL is a powerful platform to express data parallel computations on a wide range of accelerator devices which are commonly used in HPC applications [2]. It has been recognized that task-allocation, i.e. the distribution of computations onto the different available computing resources, on heterogeneous systems is an important problem [5,11].

To address this problem, schedulers for heterogeneous platforms, such as Qilin [11] or StarPU [5] have been proposed. The quality of their schedules is highly dependent on a realistic execution time prediction for the individual computations on each specific device. Measuring the execution time for all computations on all possible devices is very time and energy consuming. Therefore, static execution time prediction models have been established [3,8,10].

These models enable high quality predictions on early generation GPGPUs. However, the caches introduced into the memory hierarchy of later GPGPU generations are neglected. We have benchmarked a large set of randomly generated memory accesses on a range of current GPGPUs. Our measurements show that the accesses can be categorized into five patterns with distinct performance characteristics. Using these measurements, we extract a performance characteristic for each pattern and incorporate it into our performance prediction model.

S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 749–760, 2015.
DOI: 10.1007/978-3-319-27308-2_60

Finally, we evaluate the quality of our improved model on a large set of randomly generated OpenCL kernels. The contributions of this paper are as follows:

- We show that memory accesses can be categorized into patterns with distinct performance characteristics.
- We present a fully static OpenCL computation performance prediction model.
- We evaluate a large set of randomly generated kernels to show that our cache-aware model enables improved predictions on GPGPUs with cached memory hierarchies.

1.1 Example

A frequent application for OpenCL computations are stencil operations on two-dimensional arrays. A simple example of such a stencil computation is given in Eq. 1 which we will apply on an array of size $n * m$. The example is chosen to illustrate the effects of different access patterns on the execution time.

$$b(i,j) = a(i,j)^2 - a(1,j) \tag{1}$$

The equation uses a stencil on the input array a to compute all elements of the output array b. The complete computation is performed in the following steps.

```
1: $n_{\mathrm{WI}} = m * n$
2: $mem_{GPU}^{input} \leftarrow device.alloc(n_{\mathrm{WI}} * s_{\mathrm{WI}})$
3: $mem_{GPU}^{output} \leftarrow device.alloc(n_{\mathrm{WI}} * s_{\mathrm{WI}})$
4: copyDataToGPU($\rightarrow mem_{GPU}^{input}$)
5: $device.kernel(n_{\mathrm{WI}}, n_{\mathrm{WG}}, m, n)$
   $\Rightarrow \forall id \in \{0, .., n_{\mathrm{WI}}\}.\ sq\_mod(mem_{GPU}^{input}, mem_{GPU}^{output}, m, n)$
6: copyDataFromGPU($\rightarrow mem_{GPU}^{output}$)
```

First, memory segments for the input array and the output array, both with data type size, s_{WI}, need to be allocated on the GPU (line 2 and 3). Next, we have to transfer the data from the host system to the GPU (line 4). In line 5, we execute the kernel which applies Eq. 1 for each element of the output array. The kernel code for *sq_mod* is shown below. In order to execute the kernel, we have to specify the number of work-items (i.e. output array elements), n_{WI}, and the number of work-items per work-group, n_{WG}, which will be executed in parallel. The parameter *id*, shown in the result of the kernel execution above, represents the index of the work-item the kernel is currently applied on. It is not explicitly given as a parameter, but can be obtained during the execution of the kernel using the function `get_global_id(0)`. Finally, the results of the computation need to be copied back to the host system for further processing (line 6).

```
kernel void sq_mod(global float *matrix, global float *res,
                   unsigned int m, unsigned int n) {
  size_t current_pos = get_global_id(0);
  unsigned int current_row = current_pos / n;
  unsigned int current_col = current_pos % n;
  res[current_pos] = matrix[current_row * n + current_col]
       * matrix[current_row * n + current_col]
       - matrix[current_col];
}
```

1.2 Prediction of Kernel Execution Times

We will develop our performance prediction model by subdividing the complete computation into several steps analogous to the example in the previous section. Hence, the overall execution time of an OpenCL computation is given by Eq. 2 which adds up the time required to copy data to the device (step 4), executing the kernel (step 5) and copying the result back (step 6). Here n_{WG} holds the number of work-items in a single work-group, and n_{XU} the number of execution units, i.e. the degree of parallelism on the device. The execution time of the actual kernel (step 5) is calculated using Eq. 3. The average execution time of each of the elementary operations $W_{\text{Op}}(n_{\text{Op}}) * t_{\text{Op}}(n_{\text{WI}})$ is added to the base overhead $t_{\text{Base}}(n_{\text{WI}})$ and divided by the utilization of the GPU $U(n_{\text{WG}}, n_{\text{XU}})$. W_{Op} are functions determining how the execution time scales with the number of operations n_{Op} of the same type occurring in the same kernel. We found that for the basic operations, the execution time does not scale linearly with the number of operations of the same type (see Subsect. 2.4). The family of functions t_{Op} assigns an execution time based on the elementary expression and the number of work-items n_{WI}. The utilization function $U(n_{\text{WG}}, n_{\text{XU}})$ describes the degree of parallelization achieved by the GPU based on the number of execution units and the size of the work-group. We will discuss each cost component in the following sections.

$$t(n_{\text{WI}}, s_{\text{WI}}, n_{\text{WG}}) = t_{\text{Transfer}}(n_{\text{WI}}, s_{\text{WI}}) + t_{\text{Kernel}}(n_{\text{WI}}, n_{\text{WG}}) \tag{2}$$

$$t_{\text{Kernel}}(n_{\text{WI}}, n_{\text{WG}}) = \frac{t_{\text{Base}}(n_{\text{WI}}) + \sum_{Op \in \text{Expr.-Types}} W_{\text{Op}}(n_{\text{Op}}) t_{\text{Op}}(n_{\text{WI}})}{U(n_{\text{WG}}, n_{\text{XU}})} \tag{3}$$

2 Runtime Model

In this section, we describe the individual components of our model for the prediction of OpenCL kernels executed on a specific GPU. We will first model the costs of the data transfer, followed by the costs for the individual kernel components. Loops, conditional statements and intrinsic functions for operations such as the square root or the sine are not yet regarded in the model. For most of the measurements, we ran microkernels with the operation that was to be classified and compared its performance with another microkernel that lacked the operation but was otherwise identical. We took multiple samples for each data point, in order to ensure a standard error ratio of 0.02.

2.1 Transfer of Data to and from the Device

Most modern GPUs have a dedicated portion of memory separate from the system's main memory, to be used exclusively by them. As mentioned above, data needs to be transferred to and from that memory. In order to predict the time spent on data transfer, one has to consider two components. The first one is the bandwidth bw, which is determined by the underlying bus system, typically

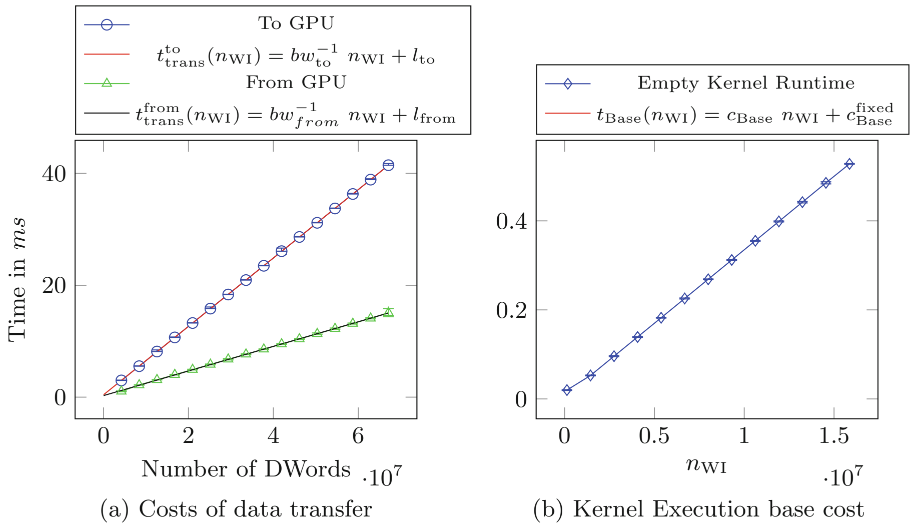


Fig. 1. On the diagram to the left, the time spent on transferring memory from and to the GPU is shown. The diagram on the right shows the base cost for kernel execution.

PCI Express, and the second one the signal propagation and access latency l_{prop}, which is determined by all the effected components, i.e. the bus systems and the memory modules [1,4,7].

We measured the time it took to transfer an array with n_{WI} work-items between system and device memory. These measurements, with their results displayed in Fig. 1a, show a linear dependency between the amount of data transferred and the transfer time. It can be seen that there is significantly more time spent copying data to the GPU than on copying the results back again, although the size of the data remains the same. This effect is also mentioned by Fuji et al. [7], who state driver optimizations as a reason for the different costs. Our model reflects this effect with separate assignments for the bandwidth bw and latency l being required for each direction [7].

2.2 Base Cost of Kernel Execution

Computations on GPUs are performed in a highly parallel fashion, with each work-item being computed in its own thread. These need to be started, coordinated and put into blocks, which causes a non-negligible amount of overhead, that, as shown in Fig. 1b, scales linearly in regards to the number of computed work-items.

2.3 Influence of the Work-Group Size

Each work-item belongs to a work-group. Work-items within a work-group are executed concurrently, and only once all work-items belonging to one work-group

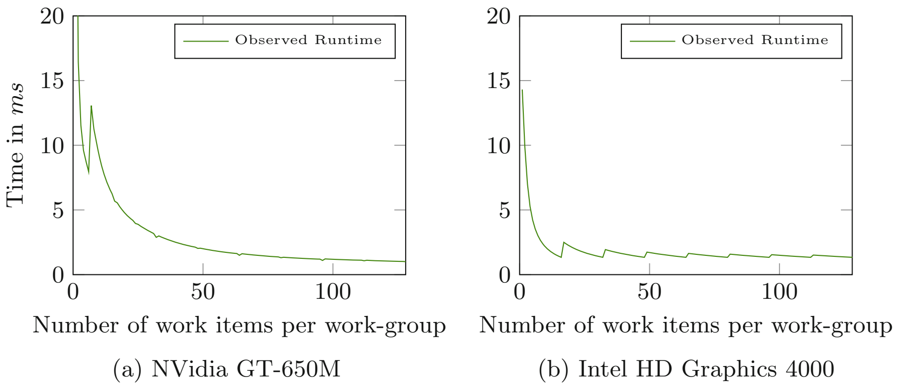


Fig. 2. Execution time for different work-group sizes. We only show the results of work-groups with less than 100 work-items in order to highlight the performance penalties of using work-group sizes that are slightly bigger than multiples of the number of execution units. The kernel we used to evaluate this behavior performs one read from and write to the global memory, and one floating point division.

are finished may work-items from another start their calculations. It follows that the work-group needs to be sufficiently large in order to fully utilize all available processing elements of the GPU. Additionally, all work-groups need to have the same size, hence the total number of work-items needs to be evenly divisible by the work-group size. However, using the biggest possible work-group size is not always optimal for maximizing GPU utilization.

We can observe the effects of improper resource utilization in Fig. 2, especially on the HD 4000. There is a clear performance degradation whenever the utilization of the execution units is less than optimal. We model the utilization as $U(w, n_{\mathrm{XU}})$, defined in Eq. 4. Term A denotes the portion of the computation that can be performed using all available execution units, while B is the remainder of the execution which does not fully utilize the available resources. Note that the size of the work-group is not determined automatically, but set by the programmer when the kernel execution is started.

$$U(n_{\mathrm{WG}}, n_{\mathrm{XU}}) = \underbrace{\frac{\left\lfloor \frac{n_{\mathrm{WG}}}{n_{\mathrm{XU}}} \right\rfloor}{\left\lceil \frac{n_{\mathrm{WG}}}{n_{\mathrm{XU}}} \right\rceil}}_{A} + \underbrace{\frac{n_{\mathrm{WG}} \bmod n_{\mathrm{XU}}}{n_{\mathrm{XU}}} \frac{\left\lceil \frac{n_{\mathrm{WG}}}{n_{\mathrm{XU}}} \right\rceil - \left\lfloor \frac{n_{\mathrm{WG}}}{n_{\mathrm{XU}}} \right\rfloor}{\left\lceil \frac{n_{\mathrm{WG}}}{n_{\mathrm{XU}}} \right\rceil}}_{B} \tag{4}$$

2.4 Basic Operations

Generally, GPU kernels consist of memory accesses and arithmetic expressions. We evaluate the behavior of arithmetic expressions, signified by the four basic operations $+$, $-$, $*$ and $/$. In order to model the execution time of basic operations, the number of work-items n_{WI} and the accumulated number of each type

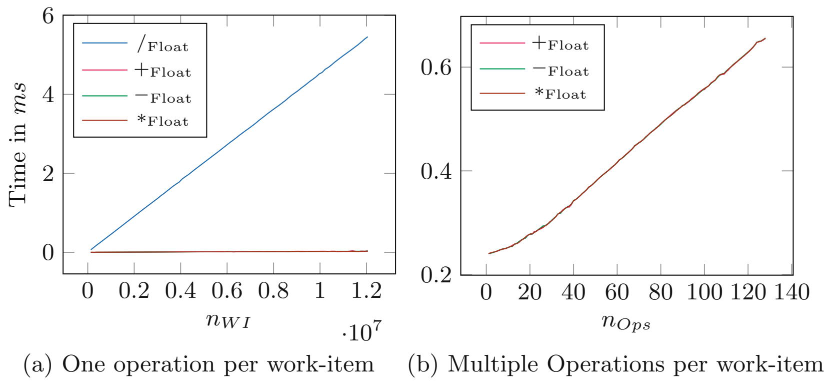


Fig. 3. The diagrams above display the progression of the execution time for basic operations, based on the number of work-items on the left and based on the number of operations in a single kernel on the right.

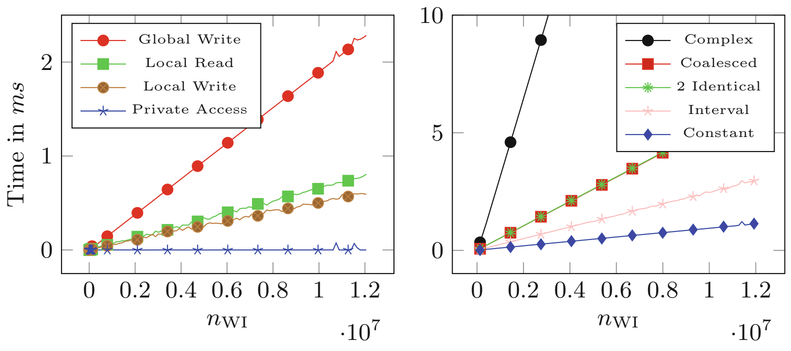
of basic operations n_{Ops} performed within the kernel must be taken into account. Our measurements (results depicted in Fig. 3a) show that there is a linear relation between the number of work-items and the execution time. For the number of operations within a work-item, we found the execution time to increase linearly as well, but only after a certain number of operations, as shown in Fig. 3b. We modeled this behavior with Eq. 5.

$$W_{\mathrm{op}}^{\mathrm{type}}(n_{\mathrm{Ops}}) = \begin{cases} a\ n_{\mathrm{Ops}}^{b} + c & : n_{\mathrm{Ops}} \leq n_{\mathrm{Ops}}^{\mathrm{sat}} \\ a' n_{\mathrm{Ops}} + c' & : n_{\mathrm{Ops}} > n_{\mathrm{Ops}}^{\mathrm{sat}} \end{cases}$$
$$t_{\mathrm{op}}^{\mathrm{type}}(n_{\mathrm{WI}}) = c_{\mathrm{op}}^{\mathrm{type}} n_{\mathrm{WI}} \tag{5}$$

The GPU dependent constants a, a', b, c, c' are obtained by fitting $W_{\mathrm{op}}^{\mathrm{type}}$ (n_{Ops}) to Fig. 3b, the constant $c_{\mathrm{op}}^{\mathrm{type}}$ is obtained by fitting $t_{\mathrm{op}}^{\mathrm{type}}(n_{\mathrm{WI}})$ to Fig. 3a.

2.5 Memory Accesses

Memory accesses scale, similarly to basic operations, linearly in regards to the number of work-items and to the number of the same access in the kernel. As expected from the three-tier memory model (with *private*, *local* and *global* memory) of OpenCL, we observed a large spread in execution time for accesses to different types of memory. In the kernel introduced in Subsect. 1.1, we see two of the different memory types in use. Private memory is implicitly used for all the parameter values and local variables. The pointers given as a parameter to the kernel point to segments of the global memory. There is one write access to the global memory, and three read accesses, with two of them accessing the same address in the same kernel execution. The third only accesses items from the first row. While local memory is not featured in the example it could be used by declaring a variable outside the kernel, e.g. `local float mem[4]`.



(a) Different kinds of memory accesses (b) Different Global Read Accesses

Fig. 4. The diagrams above display the results of our measurements for the cost of different memory accesses. In the left picture, the different kinds of memory accesses are compared to each other. On the right side, read accesses to the global memory with different access patterns are displayed. The memory accesses used for these measurements are the ones given in the text.

These are usually implemented using different memory types, which, as we concluded from the result of our measurements, depicted in Fig. 4a, differ significantly in terms of access time. Private memory accesses are practically cost-free. Local accesses, while more expensive, are still significantly faster than global accesses.

However, only distinguishing between different memory types has proven insufficient for obtaining precise estimations the execution time of kernels. In the following, we discuss groups of memory access patterns identified by our measurements which are shown in Fig. 4b. Each pattern is associated with a characteristic memory access latency, induced by the caches utilized by the respective pattern. In the patterns, `x` denotes the index of the current work-item which can be computed using $get\_global\_id(0)$ inside the kernel.

Some memory accesses always read from the same address, e.g. `matrix [0x2a]`. The content of this address only needs to be cached once. Further reads, including those made from other work-items that are able to access the same cache, do not need to fetch from the off-chip memory a second time. We refer to these as *constant accesses*.

If all of the addresses that are reached by an access are in a range that fits into the cache – the size of which may be queried from the OpenCL runtime – of that device (e.g. `matrix[x&0xFF]`), the values can still be held in the cache, and expensive off-chip accesses may be avoided. While this kind of read is more expensive than the constant one, the costs of the *interval access* are still below the costs of other global accesses.

The most common memory access ranges over a memory area that is significantly greater than the typical cache (e.g. `matrix[x]`). However, as this is the default case, the GPU vendors implemented optimizations [12], and typically several addresses will be loaded at once, reducing the number of accesses to the off-chip memory significantly. This kind of access is called a *coalesced access*.

In some kernels, there are *multiple identical accesses* to the same address (e.g. `matrix[x] * matrix[x]`). We found subsequent accesses to the same address within the same kernel to be cheaper compared to accesses to a different address. On the GT-650M, we observed subsequent reads to the same address to be without any extra cost, while for other GPUs the overhead was comparable to constant accesses. The reason for the speedups may be compiler optimizations. These values may be stored in a cache, or the compiler may store them in a register file. This eliminates or severely reduces the cost of subsequent accesses.

Accesses that do not fall into those categories, are referred to as *uncoalesced accesses*. With those, caches cannot hide the latency, and the off-chip memory needs to be accessed frequently. In consequence, these kinds of accesses are at least one order of magnitude slower than the others.

Note that for some access patterns, our analysis needs to compute the size of the input arrays statically which is not always possible.

3 Empirical Evaluation

With the model described in the previous section, we built an analysis that iterates over the syntax tree of the OpenCL kernel and collects the number of occurrences of each elementary expression. Given the kernel, the experimentally gathered constants for the GPU operations, the number of work-items n_{WI} and the desired work-group size n_{WG}, we can predict the total execution time for the computation. Figure 5a displays the analysis result for a computation using the kernel from the example in Subsect. 1.1 with an array size $m * n = n_{\mathrm{WI}}$ of 4096×4096.

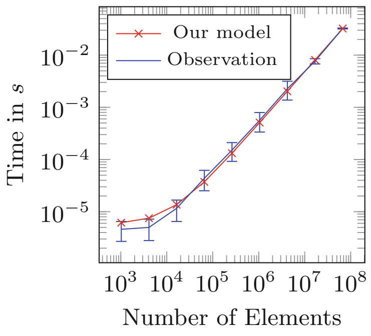
In Fig. 5b, the time spent executing the example from Subsect. 1.1 with different input array sizes is compared to predictions that were made during its compilation. For arrays with more than 128×128 entries, the observed result is very close to the predicted one. Only for smaller arrays does the model slightly over-estimate the approximation. This may be caused by the larger uncertainty in the measurements for very small arrays.

4 Quantitative Evaluation

To evaluate the quality of the predictions made with the model, we generated random valid OpenCL kernels that perform stencil computations. We created execution time predictions for each kernel, and compared them to the arithmetic mean of the run time of five different runs of the same kernel. This was done for array sizes ranging from 32×32 to 8192×8192 elements. We performed this evaluation on two thousand generated kernels, separated into two categories with one thousand samples each. Kernels in the first category have a small number of expressions and the complexity of memory access patterns is limited to simple patterns as well. Such kernels may conceivably be used in productive environments, and will henceforth be referred to as realistic samples. In the other category, the complexity of the kernels is not restricted. In order to avoid stack

Cost Type	# in Kernel	Time in μs
$-_{\text{float}}$	1	74.16
$*_{\text{float}}$	1	74.54
$+_{\text{int}}$	1	55.13
$*_{\text{int}}$	7	81.04
$/_{\text{int}}$	4	1506
private access	1	0.0
interval global read access	1	770.9
continuous global read access	1	2335
base cost	1	3191
work-group size	1024	-
final prediction		**8089**

(a) Prediction table for 2^{24} work-items (Array of size 4096×4096)



(b) Comparison between predicted and actual execution time

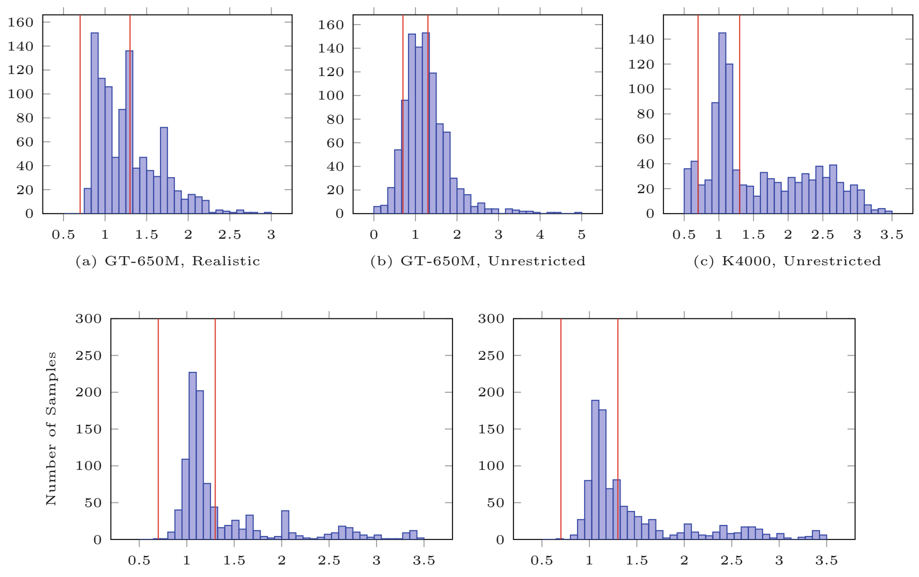
Fig. 5. This table shows the predictions for the example kernel presented in Subsect. 1.1. On the left, the individual elementary expressions are listed for a fixed work-item size on the test system, and on the right the overall predictions are compared to the actual execution time of the kernel. The prediction for this kernel may be computed as follows: $t_{\text{Kernel}}(n_{\text{WI}}, n_{\text{WG}}) = M_{\text{WG}}(n_{\text{WG}})((c_{\text{Base}} + c_{-}^{float} + c_{*}^{float} + c_{+}^{int} + \frac{7}{7}c_{*}^{int} + \frac{4}{4}c_{/}^{int} + c_{\text{global}}^{\text{ivl}} + c_{\text{global}}^{\text{cont}})n_{\text{WI}} + c_{\text{Base}}^{\text{fixed}})$. Both examples were measured on a NVidia GT-650M.

overflows during the generation, we limited the number of subexpressions to at most 50 elementary expressions for the calculation. Each generated memory access may have up to fifty subexpressions in its index expression.

The histograms in Fig. 6 display the result of this evaluation. We used the quotient $\frac{\text{prediction}}{\text{result}}$ as a measure of the quality for the result. A perfect prediction yields 1.0, values smaller than that indicate an underestimation, and larger values an overestimation. In the histograms, one may see that the prediction performed poorly in some cases. A possible reason may be an incorrect classification of memory accesses, where the actual runtime was lower than the predicted one. Nevertheless, on the Quadro K4000, we observed deviations of less than 30 % from the perfect result in realistic sample set for 71 % of the samples. For the unrestricted sample set 43 % were in that interval. The GT-650M has a slightly different behavior. Here, 63 % of the predictions for the realistic set deviate less than 30 % from the perfect result. For the unrestricted sample set, 50 % deviate less than 30 %. We also compared our model to a version that does not use our memory access patterns (see Fig. 6e). In contrast to the run that used our model, only 61 % instead of 71 % of all samples deviate by less than 30 %. This shows that the quality of the prediction can be improved by taking GPU caches into account.

5 Related Work

In the eight years since GPGPU programming emerged, there have been several approaches to modeling the execution time of programs running on the GPU. They can be grouped into two major groups, static and dynamic approaches.



(a) GT-650M, Realistic (b) GT-650M, Unrestricted (c) K4000, Unrestricted

(d) K4000, Realistic Samples, Our model (e) K4000, Realistic Samples, Simple model

Fig. 6. The diagrams show distributions of benchmark results for both the test with an unrestricted as well as with a restricted set of OpenCL kernels. We performed the evaluations on a NVidia GT-650M notebook GPU and a Quadro K4000 workstation GPU. The X axis denotes the quotient $\frac{t_{\text{prediction}}}{t_{\text{result}}}$. The closer this quotient is to 1, the better the prediction. Using the diagrams in the second row, one can compare the quality of predictions made using our model with the quality of the predictions made using a simple model that does not model cache behavior.

C. Luk et al. [11] and G. Diamos and S.Yalamanchili [6] propose dynamic approaches in order to make scheduling decisions for their respective heterogeneous environments. They do this by observing run times of kernels executed on the GPU and using the gathered information to extrapolate information about future kernel executions. One big disadvantage with dynamic approaches is the fact that they do not possess information before the kernel is executed for the first time. Without any static metric, this has a negative impact on the quality of schedules involving computations on unknown kernels. One possible alleviation of this problem to use a static approach such as ours for the initial estimation, and then use the dynamically gathered data to improve on the initial value [5,6,8,11].

D. Grewe and M. O'Boyle [8] propose a static, machine-learning based approach. They consider OpenCL kernels, and split them into components. They take a wider range of basic operations, e.g. *sqrt* or *sin* into account. However, the paper does not model different global memory accesses and their possible caching [8].

K.Kothapalli et al. [10] contribute a static performance model based on NVidia's CUDA framework. They analyze the program based on specifications provided by NVidia. Their model is more closely tied to the architecture of the NVidia hardware. They distinguish between the different kinds of memories, e.g. local and global, but do not incorporate caches into their model [10].

C. Martel et al. [3] propose a performance model that splits the execution time of a kernel into three components: computation time, time spent on transferring from global to local memory, and time spent on transfers from local to private memory. The described model is more detailed in regards to the effects of the size of the work-group, but also neglects caches.

6 Conclusion

In this paper, we presented an approach for the prediction of OpenCL kernel execution times. The model we presented takes the different subcomponents of the computation into account, and also distinguishes between the transfer of data to and from the GPU, and the computation itself. Memory accesses, which are a significant factor for the determination of the overall kernel execution time, are classified according to their cache-behavior.

For smaller OpenCL kernels, which conceivably occur in productive environments, our model is able to deliver predictions that are sufficiently accurate for use in productive applications that require estimation about the duration of tasks in heterogeneous environments. These systems may utilize the model in order to get an initial estimation of the duration of a task without the need to execute it, and enable them, in concert with other models, e.g. for CPUs or Accelerator platforms, to find a schedule that optimizes the overall utilization and performance of the system.

In future work, we plan on including language features that are currently not considered into the model, e.g. for loops, and the intrinsic functions defined by the OpenCL standard. Another topic of interest is the *Standard Portable Intermediate Representation*, which will be incorporated in the OpenCL standard in version 2.1 and may enable us to gain a more fine-granular view of the operations performed on the hardware and hence provide more elementary predictions [9].

Acknowledgments. This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

References

1. Intel X79 express chipset block diagram. http://www.intel.de/content/www/de/de/chipsets/performance-chipsets/x79-express-chipset-diagram.html
2. Top 500 list, November 2014. http://www.top500.org/lists/2014/11/
3. Alberto, C.M.M., Sato, H.: Linear performance-breakdown model: a framework for GPU kernel programs performance analysis. Int. J. Netw. Comput. **5**(1), 86–104 (2015)

4. Weiler, A., Pakosta, A.: High-speed layout guidelines. Technical report, Texas Instruments, November 2006
5. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 863–874. Springer, Heidelberg (2009)
6. Diamos, G.F., Yalamanchili, S.: Harmony: an execution model and runtime for heterogeneous many core systems. In: Proceedings of the 17th International Symposium on High Performance Distributed Computing, HPDC 2008, pp. 197–200. ACM, New York (2008). http://doi.acm.org/10.1145/1383422.1383447
7. Fujii, Y., Azumi, T., Nishio, N., Kato, S., Edahiro, M.: Data transfer matters for GPU computing. In: 2013 International Conference on Parallel and Distributed Systems (ICPADS), pp. 275–282, December 2013
8. Grewe, D., O'Boyle, M.F.P.: A static task partitioning approach for heterogeneous systems using OpenCL. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 286–305. Springer, Heidelberg (2011)
9. The Khronos Group: The OpenCL specification (provisional), version 2.1, January 2015. https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf
10. Kothapalli, K., Mukherjee, R., Rehman, M., Patidar, S., Narayanan, P., Srinathan, K.: A performance prediction model for the CUDA GPGPU platform. In: 2009 International Conference on High Performance Computing (HiPC), pp. 463–472, December 2009
11. Luk, C.K., Hong, S., Kim, H.: Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: 42nd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-42, pp. 45–55, December 2009
12. NVidia: OpenCL programming guide for the CUDA architecture. Programming Guide, September 2012. http://hpc.oit.uci.edu/nvidia-doc/sdk-cuda-doc/OpenCL/doc/OpenCL_Programming_Guide.pdf