# Future of GPGPU Micro-Architectural Parameters

Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal
Eindhoven University of Technology, The Netherlands
c.nugteren@tue.nl, g.j.w.v.d.braak@tue.nl, h.corporaal@tue.nl

***Abstract*—As graphics processing units (GPUs) are becoming increasingly popular for *general purpose* workloads (GPGPU), the question arises how such processors will evolve architecturally in the near future. In this work, we identify and discuss trade-offs for three GPU architecture parameters: active thread count, compute-memory ratio, and cluster and warp sizing. For each parameter, we propose changes to improve GPU design, keeping in mind trends such as dark silicon and the increasing popularity of GPGPU architectures. A key-enabler is dynamism and workload-adaptiveness, enabling among others: dynamic register file sizing, latency aware scheduling, roofline-aware DVFS, run-time cluster fusion, and dynamic warp sizing.**

## I. Introduction

Performance of single-core processors has shown an exponential growth for the past decades. This exponential growth ended in 2004, mainly limited by the power wall [2]. Performance growth was re-enabled through parallelism, i.e. by placing multiple processor cores on a single chip. This has led to heterogeneous computing environments, in which multi-core CPUs are used in conjunction with massively parallel accelerators. An example of such an accelerator is the graphics processing unit (GPU), originally designed for graphics rendering through OpenGL and DirectX, but used increasingly nowadays for *general purpose* compute workloads (GPGPU) through languages such as CUDA and OpenCL.

Programming of GPGPUs has become increasingly accessible, leading to an increased use of GPUs to accelerate parallel computations such as linear algebra, image processing, finance, molecular dynamics, and graph traversal [5]. Although GPUs are used for such computations in a wide range of form factors - from mobile phones to supercomputers - the GPU architecture itself is still primarily designed for its original purpose: graphics. For this reason, there are many open questions regarding GPGPU architecture design. In particular, we believe that researchers and designers working on various aspects related to GPGPUs (e.g. architecture, programming languages, compilers, applications) can greatly benefit from the identification and evaluation of high-level architectural parameters for current GPU architectures and from an outlook towards the future of GPGPU architectures. In the future, as Moore's law and technology scaling will continue, while power consumption remains limited, so-called *dark silicon* [1] will be introduced to microprocessors. A large number of power hungry transistors will only be allowed as long as a they are not all switched on together. To maintain steady growth of GPGPU performance in new designs, dark-silicon will ask for new workload-aware architecture improvements.

In this work we perform a study of high-level architectural parameters with a focus on the future of GPGPU architectures. Our contributions can be summarised as follows. We identify three GPU architecture parameters, discuss their trade-offs and provide an outlook for the future, keeping in mind trends such as dark silicon[1] and the shift towards dedicated GPGPU architectures [5]. The parameters identified in this study along with the architectural improvements proposed are as follows:

1) ACTIVE THREAD COUNT The number of threads active, a key parameter to be able to hide pipeline and off-chip memory access latencies. We propose **dynamic register file sizing** and **latency aware scheduling** in section II.
2) COMPUTE-MEMORY RATIO The ratio between the compute power (in GFLOPS) and the off-chip memory bandwidth (in GB/s). We propose **roofline-aware DVFS** and **dynamic compute cluster disabling** in section III.
3) CLUSTER AND WARP SIZING The amount of processing elements in a compute cluster[2] and the amount of threads grouped to execute together. We propose **run-time cluster fusion** and **dynamic warp formation and sizing** in section IV.

As an example throughout the paper, we use NVIDIA's G80 GPU [7], although most desktop GPU architectures have a fairly similar high-level design. The G80 architecture has up to 16 clusters, each containing 8 processing elements (PEs). PEs in a cluster share an instruction cache, instruction fetch stage, a register file, and an on-chip scratchpad memory.

## II. Parameter 1: Active thread count

One of the main characteristics of a GPU is its ability to hide pipeline and off-chip memory latencies through zero-overhead thread switching. If sufficient parallelism is exposed and the GPU's register file is large enough, these latencies can be hidden completely. The main benefits of such an architecture are: 1) the ability to use special high-throughput high-latency off-chip memory (e.g. GDDR5), 2) caches are no longer required to hide long access latencies, 3) simplification of the micro-architecture: there is no need for out-of-order superscalar execution, and 4), the ability to use a long pipeline.

A GPU needs a large register file to hide latencies. This register file will be able to store the context for each of the so-called *active threads*, enabling zero-cycle context switching. For example, for the G80 with 8 PEs per cluster, the pipeline latency for basic ALU operations is 22 cycles [10] and the

[1]Dark silicon comes at an increased design complexity cost, see also [1].
[2]Equivalent to NVIDIA's multiprocessor (SM) and AMD's compute unit.

off-chip memory access latency is on average 600 cycles [10]. If we assume that instructions in a thread are dependent on the previous instruction, we need at least $22 \cdot 8 = 176$ active threads per cluster to hide the pipeline latencies. With off-chip loads only, we need $\pm 600 \cdot 8 = \pm 4800$ or more active threads. In reality, 256 or 512 active threads per cluster on the G80 is for most realistic workloads enough: not all instructions are memory accesses and dependent on the previous instruction.

### A. *Trade-offs*

Allowing a large active thread count requires a large register file, consuming power and occupying chip area. However, when the active thread count is not high enough to hide pipeline and off-chip memory access latencies, performance can drop significantly (proportional to the number of absent threads in the worst case). The optimal value for this parameter is dependent on many factors: the pipeline depth, the off-chip memory characteristics, the amount of PEs per cluster, the thread scheduling mechanism, and the workload.

Because the workload is typically unknown at processor design time, dealing with this trade-off is a dynamic problem. Performing a detailed workload analysis might help to get valuable insight through the examination of distributions of occurrences of off-chip loads and dependencies in typical workloads. However, the behaviour of threads is difficult to predict, as they do not execute in lock-step, but rather diverge and arrive at different instructions at different times.

### B. *Outlook*

As an outlook towards the future of GPGPU architectures, we propose two techniques to improve performance and energy efficiency with respect to the number of active threads: dynamic register file sizing and latency-aware scheduling.

With dark silicon reaching GPUs in the future, we propose the addition of a **dynamically-sized register file** to each cluster. To determine which part of the register file is required to accommodate a sufficient amount of active threads, instruction sequences (e.g. at kernel-level) can be analysed statically where possible and dynamically otherwise. Switching the complete register file on will only be possible if another component is switched off, e.g. a cache. Such a register file will benefit from power savings (as shown for a hierarchical register file [4]) for workloads that only require a low amount of active threads, in contrast to current GPUs, which greedily run the highest possible amount of threads.

Furthermore, we believe that an improved **latency-aware scheduling** algorithm can reduce the required amount of active threads. The thread-scheduler in the G80 architecture (and in e.g. the GT200 and Fermi architectures) schedules groups of threads (*warps*[3]) that are ready for execution in a round-robin fashion, instruction per instruction [10]. For example, in Listing 1, instruction 1 will first be executed for all threads, followed by instruction 2, 3, and 4. However, since instruction 4 is dependent on the result from the off-chip load, a large number of active threads is required to hide the latency of instruction 3. In contrast, a latency-aware thread scheduler would give preference to scheduling instructions 1, 2 and 3 for a subset of the active warps first, such that the long latency of instruction 3 can be hidden by executing instructions 1, 2 and 3 of a second subset of active threads. To hide the pipeline latencies as well, the subset size must be set equal or larger to the pipeline latency. A related idea (two-level warp scheduling) shows potential: performance is increased by 19% [9].

```
instruction 1:        $r0 ← 2
instruction 2:        $r1 ← $r0 * 4
instruction 3:        $r2 ← load[$r1]
instruction 4:        $r3 ← $r2 * $r2
```

Listing 1. Example pseudo-assembly GPU kernel code.

## III. Parameter 2: Compute-memory ratio

The GPU architecture is known to be well-suited for throughput-oriented applications [2]. This is accomplished by two means: 1) by providing a large number of high clock frequency processing elements, and 2), by providing a high bandwidth to off-chip memory. While the first ensures a high instruction throughput, typically measured in giga-floating point operations per second (GFLOPS), the second enables a high data throughput, measured in gigabytes/s (GB/s). The ratio between the two (GFLOPS versus GB/s), the *compute-memory ratio*, is an important design parameter for GPUs.

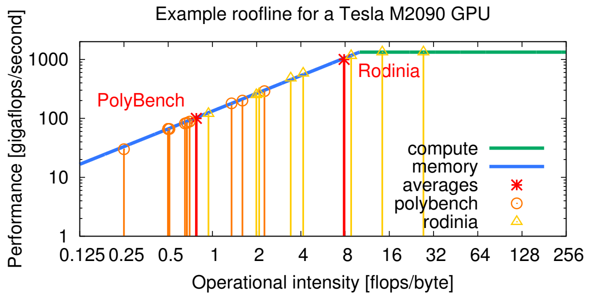

Fig. 1. Roofline with several data points for a Tesla M2090 GPU.

Applications can either be *compute-bound*, i.e. limited by the peak instruction throughput, or *memory-bound*, i.e. limited by off-chip memory bandwidth. The metric *operational intensity* (measured in operations per byte) determines which limit applies. We have collected operational intensities for two benchmark suites in Table I: PolyBench/GPU and Rodinia[4]. We note that benchmarks from PolyBench/GPU are overall much smaller and more memory intensive, while Rodinia benchmarks are more compute intensive. To visualise the limitations of the compute-memory ratio, the *roofline model* was introduced [11]. We give an example of the roofline model for a Tesla M2090 GPU in Fig. 1, in which we also plot the average operational intensities from the results of the two suites from Table I. The roofline model shows the maximum achievable performance of a specific application, which is limited by one of the two bounds.

[3]The term *warp* as used throughout this paper is NVIDIA specific, AMD refers to the same concept with the term *wavefront*.

[4]PolyBench/GPU can be found at www.cse.ohio-state.edu/~pouchet/software/polybench/, Rodinia at www.cs.virginia.edu/~skadron/wiki/rodinia/.

TABLE I
OPERATIONAL INTENSITIES FOR THE POLYBENCH/GPU (2DCONV - SYRK) AND RODINIA (BACKP - PATH) BENCHMARK SETS.

| | **2dconv** | **2mm** | **3dconv** | **3mm** | **atax** | **bicg** | **corr** | **covar** | **fdtd-2d** | **gemm** | **gesummv** | **gramsch** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Oper. int. [flops/byte]** | 1.35 | 0.50 | 1.60 | 0.51 | 0.50 | 0.50 | 0.50 | 0.50 | 2.26 | 0.67 | 0.25 | 0.65 |
| **Limit on Tesla M2090** | mem | mem | mem | mem | mem | mem | mem | mem | mem | mem | mem | mem |
| | **mvt** | **syr2k** | **syrk** | **—** | **backp** | **bfs** | **gauss** | **hotspot** | **kmeans** | **nw** | **particle** | **path** |
| **Oper. int. [flops/byte]** | 0.50 | 0.70 | 0.67 | — | 3.41 | 0.94 | 2.08 | 27.33 | 4.15 | 8.79 | 1.99 | 14.33 |
| **Limit on Tesla M2090** | mem | mem | mem | — | mem | mem | mem | comp | mem | comp | mem | comp |

### A. Trade-offs

To increase the peak instruction throughput, a GPU designer can add more processing elements, increase their clock frequency, or increase the IPC of individual processing elements. On the other hand, to increase the peak off-chip memory bandwidth, a GPU designer can either increase the clock frequency of the memory, or increase the bus-width between the memory and the processor itself. Again, these design choices are a dynamic problem: they have to be made based on information from the workload. Performing a detailed workload analysis will help towards solving the compute-memory ratio trade-off for the average case, but will still result in over-dimensioned hardware in practice: the variance among applications (see Table I for example) is too large.

### B. Outlook

For future GPGPU architectures, we need to create a dynamic compute-memory ratio (or: roofline) to improve the overall power efficiency. This dynamic roofline can either be discrete, i.e. the GPU switches to fixed operation points, or continuous. We distinguish two techniques to create a dynamic roofline: roofline-aware DVFS and a dynamic cluster count, and make a note towards increasing the operational intensity.

Dynamic frequency scaling can be used to reduce the clock frequency of the GPU core or off-chip memory, which has a linear impact on either the compute or the memory roofline. For example, most benchmarks from PolyBench/GPU (see Table I) allow halving the compute frequency without loosing performance[5], while saving a factor of two in terms of power. Furthermore, dynamic frequency and voltage scaling (DVFS) might be applied to lower the voltage as well as the frequency for cubic gains in power ($P = \alpha \cdot f \cdot C \cdot V^2$). We therefore propose a **roofline-aware DVFS** scheme to create a dynamic roofline, saving power while maintaining performance.

Similarly, linear power gains can be obtained by temporarily powering down complete clusters of PEs for memory-bound workloads. **Dynamic compute cluster disabling** is enabled by the GPU's modular architecture and can be applied at kernel-granularity. Again, such a technique lowers the compute roofline, saving power without compromising performance.

Furthermore, we also identify the need to architecturally **increase the operational intensity** of applications for future GPGPUs. An increasing amount of GPU kernels will hit the *memory wall* in the near future: the growth of compute performance is predicted to outgrow memory bandwidth growth, despite emerging technologies such as 3D-stacking [5]. To maintain performance growth, future architectures will need to improve data-locality by increasing sizes of register files, scratchpad memories, and caches.

[5]It should be noted that the maximum bandwidth might not be achieved for low core clock frequencies due to a too low memory request rate.

## IV. PARAMETER 3: CLUSTER AND WARP SIZING

Processing elements in GPUs are typically clustered into smaller groups. The G80 architecture for example can scale up to 16 clusters, with each cluster containing 8 PEs. On each cluster, instructions are executed as *warps*: groups of threads executing in lock-step. In Table II we list several NVIDIA GPUs along with their cluster and warp sizes. We identify the cluster and warp sizes as our third GPU design parameter.

TABLE II
WARP AND CLUSTER SIZES FOR VARIOUS NVIDIA GPUS.

| | Cluster size | Warps issued per cluster per cycle | Warp size | Release year |
|---|---|---|---|---|
| G80 (Tesla) | 8 | 1 | 32 | 2006 |
| GT200 (Tesla) | 8 | 1 | 32 | 2008 |
| GF100 (Fermi) | 32 | 2 | 32 | 2010 |
| GF104 (Fermi) | 48 | 4 | 32 | 2010 |
| GK104 (Kepler) | 192 | 8 | 32 | 2012 |
| GK110 (Kepler) | 256 | 8 | 32 | 2013 |

We define a (SIMD) cluster as a set of PEs that share one or more common components (e.g. a memory, a pipeline stage) that are inaccessible by the remaining PEs in the GPU. For programming models such as CUDA and OpenCL, a *threadblock* or *workgroup* typically has to map in its entirety on a single cluster in a GPU. Within a G80 cluster, PEs share among others an instruction cache, an instruction fetch and decode stage, a scratchpad memory, and a texture cache.

Since PEs in a cluster share the first few pipeline stages with other PEs in the G80 architecture, they are required to execute the same instruction at the same time. A cluster thus forms a natural fit to execute instructions from a single warp. Nevertheless, in typical GPU architectures (as shown in Table II), the warp size and the cluster size do not necessarily match. In fact, on the G80, the 8 PEs in a cluster schedule the execution of a single warp in time, i.e. using 4 clock cycles [7]. In this way, warps are used to create virtual clusters in time (32 PEs for this example). Energy is saved by creating two clock domains, which is possible since only a single instruction needs to be fetched and decoded every 4 cycles. Newer GPUs complicate cluster design slightly, as they are able to issue multiple warps per cluster per cycle [10].

### A. Trade-offs

In order to highlight the trade-offs of cluster and warp sizing, we discuss two extremes: a cluster size equal to the total amount of PEs, and a cluster size of 1. For clarity, we assume in these cases a warp size equal to the cluster size and a single clock domain.

Many advantages can be found when designing a GPU as one large cluster (e.g. a cluster size of $16 \cdot 8$ for the G80). If we take the G80 cluster design as a starting point, we find the following main advantages: 1) a significant area reduction and power saving by sharing various stages of the pipeline (e.g. instruction fetch, instruction decode, branch logic), 2) a single (larger) scratchpad memory can replace the existing smaller memories, 3) more inter-thread communication is possible through the execution of potentially larger threadblocks or workgroups, and 4), an increase in coalesced memory accesses by recombining threads as detailed in [6]. At the other extreme is a GPU architecture for which every cluster has only a single PE. The main advantages for such a configuration are: 1) every PE can execute independently from the others, i.e. there is no branch divergence penalty, and 2) a cluster takes a small fraction of the total chip area, making routing (e.g. of the clock tree) relatively easy.

The advantages for each of these cases become automatically disadvantages for the other. This creates a trade-off with many factors, some of which are dynamic: an optimal value can only be determined at run-time. The warp size can furthermore be adjusted to create virtual clusters, sharing many of the same trade-offs. However, instead of reducing the chip area when creating larger clusters, a larger warp size will allow a reduction in clock frequency of the first pipeline stages, as only a single instruction needs to be decoded per warp.

### B. Outlook

We propose two techniques to address the trade-offs for cluster and warp sizing for future GPGPU architectures: run-time cluster fusion and dynamic warp formation and sizing.

Because a large cluster size has many advantages, namely in terms of area and energy savings, it is appealing to design a GPU with such a configuration. However, such a large cluster can result in a severe penalty in case of divergent workloads: one or more orders of magnitude depending on the amount of PEs and the workload. To still be able to accommodate such workloads, we propose to split a larger cluster in several smaller clusters at run-time, creating an adaptive configuration which can be changed at kernel-granularity (through performing static analysis or profiling). To be able to enable **run-time cluster fusion**, the hardware needs to be able to accommodate the smallest cluster size and thus include for example an instruction fetch and decode stage for every cluster. Dark silicon will however justify these additional area costs by power gating these components when the GPU is configured for non-divergent workloads. Overall, a large cluster will significantly improve power efficiency for non-divergent workloads by saving power and increasing performance through creating for example increased opportunities for memory coalescing.

Secondly, we believe future GPGPU architectures will allow **dynamic warp formation and sizing**. On current GPUs, warps are formed statically based on thread indexing. However, several works have proposed to re-combine warps at run-time [3], [8], [9], either to improve memory coalescing or to reduce branch divergence. Apart from dynamic warp formation, warp sizing can also play a major role to improve energy efficiency. Warp sizing is further discussed in [6].

## V. Related work

To the best of our knowledge, this is the first work to present an overview of GPGPU design parameters and an outlook towards the future. As for the presented ideas, 3 out of our 6 proposals have related work in literature: a hierarchical register file is presented in [4], thread scheduling in [4], [9], and warp sizing and formation in [3], [6], [8], [9].

## VI. Conclusions

In this work, we presented an overview of GPGPU architecture parameters, identified trade-offs, and gave an outlook towards the future of GPGPU design, keeping in mind trends such as dark silicon. We identified and discussed trade-offs for three architecture parameters: active thread count, compute-memory ratio, and cluster and warp sizing. For each parameter, we propose changes to improve current GPU design drastically in terms of power and performance. The improvements as discussed include: dynamic register file sizing, latency aware scheduling, roofline-aware DVFS, dynamic compute cluster disabling, run-time cluster fusion, and dynamic warp sizing. Common to these improvements is the dynamism and workload-adaptiveness of the architecture: future GPGPUs will need to set the main architectural parameters at run-time to ensure a high power efficiency.

## References

[1] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *ISCA '11: 38th International Symposium on Computer Architecture*. ACM, 2011.

[2] S. H. Fuller and L. I. Millett. Computing Performance: Game Over or Next Level? *IEEE Computer*, 44:31–38, January 2011.

[3] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *MICRO '07: 40th International Symposium on Microarchitecture*. IEEE, 2007.

[4] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron. A Hierarchical Thread Scheduler and Register File for Energy-Efficient Throughput Processors. *ACM Trans. Comput. Syst.*, 30:8:1–8:38, 2012.

[5] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31:7–17, 2011.

[6] A. Lashgar, A. Baniasadi, and A. Khonsari. Dynamic Warp Resizing: Analysis and Benefits in High-Performance SIMT. In *ICCD: 30th International Conference on Computer Design*. IEEE, 2012.

[7] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Arch. *IEEE Micro*, 28:39–55, 2008.

[8] J. Meng, D. Tarjan, and K. Skadron. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *ISCA '10: 37th International Symposium on Computer Architecture*. ACM, 2010.

[9] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *MICRO-44: International Symposium on Microarchitecture*. ACM, 2011.

[10] NVIDIA. *CUDA C Programming Guide 4.2*, 2012.

[11] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52:65–76, April 2009.