# Data placement on GPUs: a literature study

Matthijs Jansen
*Computer science UvA / VU*
Amsterdam, The Netherlands
11045663 / 2655671

***Abstract*—The Graphics processing unit (GPU) is a throughput-oriented accelerator which has seen a rapid evolution in computing power over the last decade [1]. While compute throughput is rapidly increasing, memory speed has barely improved. This is the so-called "memory gap". A lot of research has been invested over the past years in bridging this gap, because it is a bottleneck for many data-intensive applications. Although a common approach to bridge the gap is to build a memory hierarchy, efficient data placement in the different layers of the memory hierarchy (i.e., placement that leads to performance improvements) remains a challenge for GPUs.**

**In this context, this literature study systematically analyses work about data placement on GPUs, examines which subtopics are missing in the landscape of papers, and hypothesises on why these are missing. We find that the topic of data placement on GPUs has been looked at from multiple angles: from software solutions which optimize kernel code to propositions for new hardware architectures which should reduce the memory gap. From this landscape, we dive into details on the specifics of compilers and runtime systems which aim to automatically optimize data accesses in GPU kernels, because these approaches have seen the most development over the last years. We present several proposed techniques, successful for different applications, but find no overarching solution that has been generally adopted in practice.**

***Index Terms*—GPGPU, Data Placement, Memory, Cache, Compiler, Runtime system**

## I. Introduction

The General Purpose Graphics Processing Unit (GPGPU) has become the most popular accelerator specialized in processing large amounts of data, be it of a graphical nature or not. While the increase in computation power of these systems still follows Moore's law, the speed of memory components cannot keep up [1]. This memory gap makes it difficult, especially for many-core systems like GPUs, to take benefit of all available computational power, because not enough data can be moved to keep all threads running. Tab. I shows this gap: while the processing power has increased by a factor of 3.4 between the Nvidia GTX 480 and GTX 980, the increase in memory bandwidth is only 27% .

Since the development of memory subsystems cannot keep up with that of their computational counterparts, the need arises for more efficient data placement to bridge the memory gap. This literature study focuses on data placement research on GPUs, to answer the question *if the memory gap can be bridged by using techniques to fully utilize the available memory hierarchy*. The study focuses exclusively on GPUs, with no attention paid to the connection between the CPU and GPU, for instance, which we see as a different part of the challenge of getting enough data on the GPU (i.e., it is more of a networking/communication-related topic, and thus out of the scope of our analysis).

The literature study is constructed as follows. First, starting from the proposed research question, we selected representative papers which match our criteria (see Section III). We then cluster the papers based on common topics, because data placement on GPUs is too broad a term. Next, we select a specific cluster (i.e., the one containing the most interesting approches) and we analyze all the cluster's papers in-depth, to discover what they contribute to the improvement of the memory subsystem's performance (see Section IV). Finally, we compare these different approaches, as presented in the papers, to demonstrate how their ideas relate to each other and what kind of research may still be needed or is missing. Ultimately, we aim to create an overview of the landscape of successful approaches on this research topic (see Section V).

## II. Background

Since GPUs are fundamentally different from CPUs, we provide an overview of the GPU memory subsystem and related components, especially since this information is needed to understand the analysis of the core papers. Note that we assume a basic understanding of CPU and GPU architectures.

We use the Fermi microarchitecture from Nvidia as an example, because it was the most used architecture in the selected papers as will be explained in the Selection section. An overview of the Fermi microarchitecture can be found in Tab. I and Fig. 1. We use the Nvidia GPU vocabulary instead of the AMD version, thus we talk about *CUDA Cores* instead of *Stream processors*. The two most relevant parts of this architecture, in the context of this study, are the thread and memory hierarchies. The thread hierarchy consists of a small number of Streaming Multiprocessors (SM) which each contain 32 CUDA Cores. A CUDA Core is the smallest computational unit, and can handle one thread at a time. Besides these 32 CUDA Cores, a SM also contains, among other components, an instruction cache, a register file, a level 1 cache, and a texture cache. All SMs share a level 2 cache and global memory.

Another advantage of GPUs compared to CPUs, besides the higher core count, is context switching. This is very expensive on the latter because of register and memory management, but because GPU threads are very lightweight, a lot less registers

[1]A similar trend is visible for multi-core CPUs, too

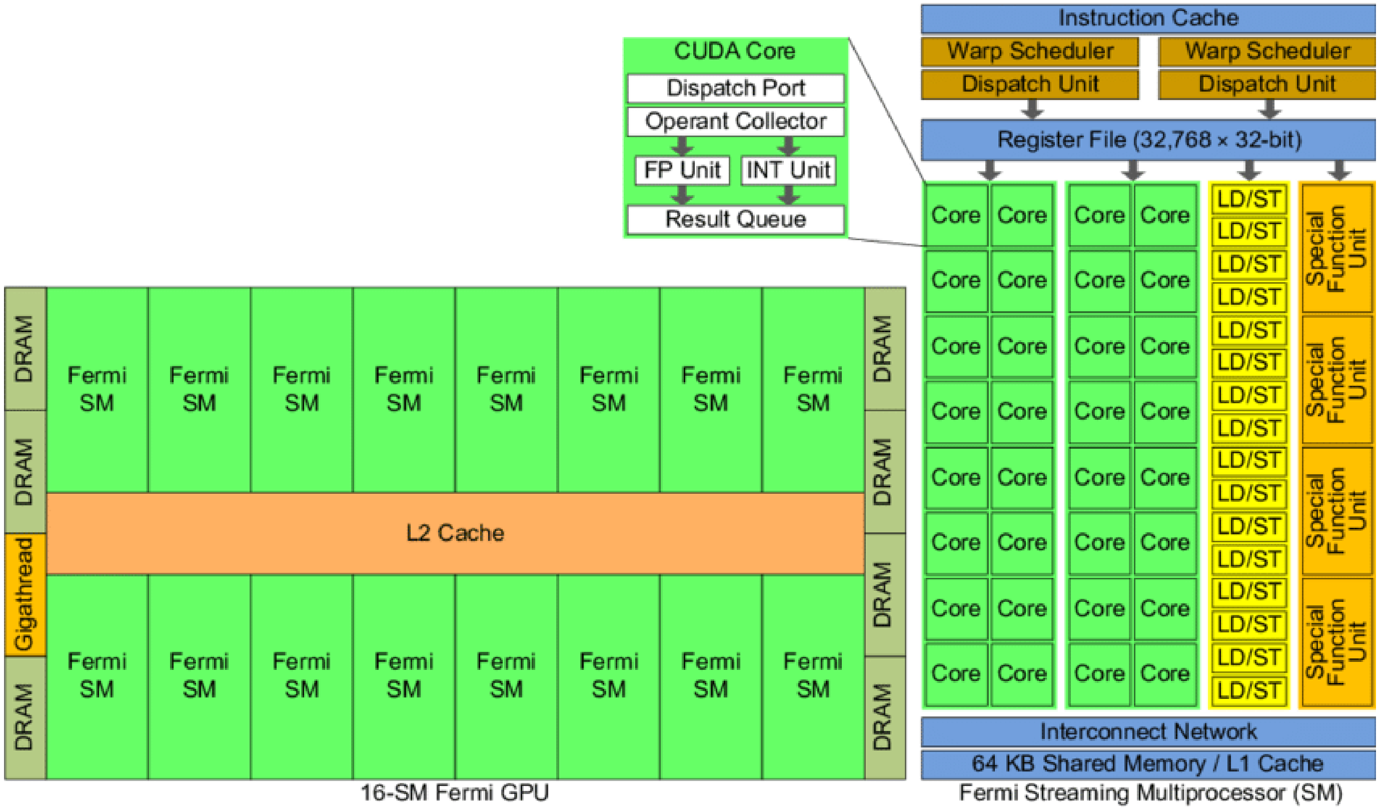

Fig. 1. The NVIDIA Fermi architecture.

and memory management has to be done. Furthermore, all this information is saved in the per-SM register file, leading to a very inexpensive context switch, and thus a higher thread-level parallelism (TLP) compared to CPUs. If the number of registers needed exceeds the size of the register file, registers will be spilled to the slower level 1 cache.

The level 1 cache and shared memory have a total size of 64 KB and can be configured in two ways: 48 KB of level 1 cache and 16 KB of shared memory, or 16 KB of level 1 cache and 48 KB of shared memory. The level 1 cache has the same function as on a CPU, while the shared memory can be used to share data between threads in a block (for instance, a part of a matrix can be stored in shared memory for efficient matrix multiplication). The shared level 2 cache is slower to access, but has a larger storage capacity. It is used to cache all reads and writes to global memory, and so is used just like the lowest-level cache (LLC) on a CPU. The same holds for global memory: here data is stored when it arrives from the PCIe-bus.

TABLE I
A COMPARISON OF HARDWARE CHARACTERISTICS ACROSS DIFFERENT GPU GENERATIONS

| | Fermi GTX 480 | Kepler GTX 680 | Maxwell GTX 980 |
|---|---|---|---|
| Processing power (GFLOPS) | 1345 | 3090 | 4612 |
| Memory bandwidth (GB/s) | 177 | 192 | 224 |
| Shared memory size (KB) | 48 | 48 | 96 |
| Register file size (KB) | 128 | 256 | 256 |

This memory hierarchy results in a request path similar to a CPU: a level 1 cache contains the data of one SM; if data is not present there, the level 2 cache will be accessed and if the data is also not present there, the global memory will be accessed. This applies for load, store, and texture operations. Texture data can be cached per SM on a special chip and on a read-only part of the level 2 cache [2]. The same applies for constant memory which is much faster than using normal caches if many threads need to read the same values; however its storage capacity is very limited.

Over time, the memory subsystem of Nvidia GPUs has changed a lot with the introduction of newer microarchitectures like Kepler and Maxwell. A detailed explanation about their memory hierarchy can be found in the papers discussed in Section IV, when applicable.

## III. Paper Selection

Hardware manufacturers like Nvidia and AMD have created many GPUs over the years, resulting in many different architectures. This poses a problem for our study, because the research topic is very hardware dependent, meaning that the same research done one year apart could result in completely different conclusions because the underlying hardware is completely different. An ideal selection criteria for finding papers for this study would be that all papers should be published around the same time and use the same architecture generation. However, there is not much research available on this topic to begin with, so this restriction would be fatal. Instead, the selection is based on how many papers could be found on the one hand, and the papers not being too outdated on the other. We performed a preliminary search, and found that most relevant papers for our topic are published in 2012 or later; therefore, we set a strict limit to only discuss papers from 2012 or younger. This means that the oldest architecture looked at from Nvidia's side is Fermi, which was released in 2010. We will not go into detail about AMD's platforms and/or solutions, because all considered papers only used Nvidia GPUs.

There are more selection criteria implemented to guarantee that the papers are coherent and can be compared later on. Firstly, the research topic, i.e., data placement on GPUs, must be the main topic of the paper, not part of a bigger subject. Secondly, the paper must include an implementation of an algorithm or a technique, presented together with benchmarked results; Ideally, the algorithm or technique should be applicable to a wide array of problems, a feature we extract from each paper's evaluation procedure and results. Finally, the paper must include a very detailed analysis of one part of the memory hierarchy, or a more high-level comparison between different parts.

Now that all search criteria have been established, we use academic search engines like *Google Scholar* and *Catalogue-Plus* to find papers. During this exploration phase, we divide papers into two categories: *Core*, which includes papers that adhere to all search criteria, and *Related*, which are papers that do not fully align to the rest or the core papers, or do not adhere to some of the criteria, like publication date, but are still interesting. We further analyse in dept the papers from the "Core" class, while the "related" papers are only subject to a quick overview.

## A. Core papers

Current research into GPU memory behaviour and optimizations is very diverse for several reasons. For one, the field of GPU research is relatively new compared to that of the CPU, so less 'best practices' have been developed, i.e., it is know well-established what is good for GPU memory-performance in general. This situation is related to GPU manufacturers, like Nvidia, not publishing critical information about hardware or scheduling policies because of competition from other manufacturers. In turn, this lack of information makes it harder for researchers to reason about performance. Furthermore, the GPU memory hierarchy and the programmer's expectations of the GPU memory system are different, since GPUs are most often used for processing large amounts of data, while this is not the general case for CPUs, the architecture with which more computer scientists are familiar.

During the selection process, we found several topical categories of papers, but only one category contains enough papers for an in-depth comparison. These are papers which use compilers to transform source code or runtime systems to change data placement decisions for better GPU memory access [3] [4] [5] [6] [7] [8], and they are the focus of this study. This imbalance is indicative of the landscape of GPU optimization research: not much research on the same topic exist, with the exception of applications studies (outside of the scope of this study) and runtime systems/compiler, which provide standard ways to automatically optimize applications for hardware. Focusing on only one core topical category has the advantage that an in-depth comparison can be done. However, with this selection, we only discuss the GPU memory placement topic from one specific viewpoint, which may be a disadvantage when wanting to get a complete overview of the relevant research landscape.

## B. Related papers

Before analyzing all core papers in-depth, we briefly discuss a few other topical categories related to GPU data placement. These categories complement the previously mentioned core papers by providing different views on the data placement subject.

The most important group of papers after the compilers and runtime systems group is hardware focused, and tries to control or modify the lower levels of the memory hierarchy for better thread-level parallelism, shared memory utilization, and memory coalescing [9] [10] [11]. These papers try to change the behaviour of the hardware so it is better fitted for most common applications, which is in contrast to the core papers which try to fit the application on the hardware by using algorithms to change the application's behaviour. Too few papers were found in this category, probably because it is a more specialized research topic than compilers and runtime systems. Reading these papers may give a better understanding of GPU data placement, especially because they complement the core papers very well.

The second category covers thread granularity and TLP. Unkule et al. [12] notice that inefficient usage of registers by kernels may result in register spilling to the slower level 1 cache, resulting in a decrease in performance. By analysing register pressure, thread granularity, data locality in thread blocks, and occupancy, kernels can be restructured to achieve better register usage while maintaining occupancy and data locality, with the goal of improving performance. Yang et al. [13] on the other hand focus on adjusting the number of threads used per code section of a kernel instead of the current standard of using the same amount of threads all the time. This will reduce the number of threads active concurrently which improves TLP and so performance. These two papers focus more on the thread hierarchy and its interaction with the memory hierarchy, which shows that this part of the architecture is as important for achieving high performance as optimizing the memory hierarchy.

The third category is about performance modeling, and is recommended to people who want to know more about performance engineering on GPUs, because the research provides a high-level overview of performance pitfalls not found on CPUs. Huang et al. [14] have created a model which predicts the performance impact of different data placement schemes on heterogeneous memory systems found on GPUs. Lai et al. [15] have analysed what performance can still be gained for matrix multiplication applications compared to the theoretical computational upper bound of GPUs, and which problems are stopping those applications from reaching the peak performance.

The final category extends the core category of compilers and runtime systems. These papers are too old (before 2012), but can still provide useful insights. Baskaran et al. [16] introduce a compiler framework based on the polyhedral model to optimize loop intensive kernels, focusing on achieving better memory access patterns. Yang et al. [17] and Jang et al. [18] use the same kind of compiler techniques to optimize loop bodies. On the other hand, Zhang et al. [19] have created a runtime system which can detect and remove dynamic irregular memory references and control flows using data reordering and job swapping.

Three papers could not be grouped into a category, but are still interesting to read and relevant for the topic at large. Yang et al. [20] implement a dynamic approach for shared memory allocation, only allocating memory when data is needed instead of allocating it for the entire lifetime of a thread block. This should increase utilization of shared memory and increase performance for kernels which heavily rely on that. Wang et al. [21] examine if different hardware for the memory subsystem can be used, hoping that using other technologies in the memory subsystem, more suited for massively parallel GPUs, will increase the performance of the whole memory subsystem. Finally, Rogers et al. [22] test a new thread scheduler which takes data locality into account, resulting in better memory usage and less thrashing in the level 1 cache. This reduces the number of slower level 2 cache accesses, giving better overall performance.

## IV. GPU DATA PLACEMENT TECHNIQUES FROM COMPILERS AND RUNTIME SYSTEMS

A compiler translates source code, in this case Nvidia's CUDA language, to a target language, for example an executable binary. However a compiler can do much more than only translating, namely analysing source code and creating target code which is more optimized in general and for certain hardware in specific. This is where most of the following papers operate: Extending basic CUDA compilers with algorithms targeted at GPU memory optimizations. The advantage of using such a compiler is clear: A programmer does not have to worry about spending many hours optimizing a kernel, the compiler can do that automatically. An even bigger advantage is that those programmers do not need to have in-depth knowledge on GPU architecture, only the creator of the compiler has to. Runtime systems differ from compiler in the way that they do not interact with an application before execution, but during. The advantage is that problems of dynamic nature, like data placement, are most of the time not known before the execution of an application and can thus only be tackled during runtime. The disadvantage is that this produces an overhead during execution, so these systems need to be kept as simple as possible. This is unlike compilers, since a long compile time is most often not a problem.

### *A. Automatic data placement into GPU On-Chip Memory Resources [3]*

A big problem in GPU performance optimization research is that very often radical differences in hardware characteristics between GPUs from different generations occur. This makes performance portability very hard since most kernel optimization is currently done by tweaking code by hand. Li et al. [3] have created a source-to-source compiler which creates performance portable code, taking the specifications of the underlying hardware into account. Now source code does not have to be changed when wanting good performance on different GPUs since the compiler can automatically do this. This research does not focus on basic optimizations like tiling and assumes that the kernels have already been optimized up to that point, either by hand or by using other compilers. Furthermore only on-chip memory is considered, so the register file, level 1 cache and shared memory. The goal of the compiler is to move data between these on-chip memory types to get faster access times and less memory pressure, resulting in a higher TLP. Between these 3 types of memory, data can be moved in 6 different ways. Since the register file is bigger than the level 1 cache and shared memory combined, moving data from the register file is ignored. Moving data to shared memory is left for future work since this involves heavy code changes. What is left is moving data to the register file and from shared memory to the level 1 cache.

The compiler applies these transformations step by step. First, the compiler investigates moving data from shared memory to registers. The register file is quicker to access, bigger and mostly underused in shared memory focused applications. Only private data is moved from shared memory since registers should not be used for sharing data, although that is possible since Kepler using warp shuffles. Second, moving data from shared memory to local (level 1 cache) or global memory. This is done if the shared memory pressure is very high, which can limit TLP. This can be done for dynamic memory accesses, since these can not be moved into registers, or for large arrays, since global memory is much larger then on-chip memory. Although the memory accesses do become slower, achieving higher TLP makes up for that. In contrast to moving data to registers, local and global memory are visible to all threads so all data from shared memory could be moved there. In the last step data movement from the level 1 cache and shared memory to registers using register tiling is analyzed. This occurs when multiple threads access the same data, which can potentially result in multiple accesses to global memory. Normally the level 1 cache would be used to cache the data after the first access but this can not be guaranteed, so the compiler explicitly moves the data to registers. The only problem is that registers are private, and can at most be shared by threads in one warp. Thread compacting has to be applied, where multiple warps or threads get compacted into one warp or thread, for this technique to work.

Having a possibility to execute one of these steps does not mean higher performance can be gained automatically. An auto-tuning phase is used for each compiler pass to determine which data movement is beneficial for performance. For example moving data from shared memory to local or global memory can increase TLP, however this could also create network contention which can adversely affect performance. To keep the cost of auto-tuning limited, the most referenced variables are tuned first. This is done incrementally, so when a certain data movement would hurt performance, the auto-tuning process stops for that phase. This algorithm is much simpler than genetic or machine learning variants that have been used in other studies but it is at least as effective and much faster.

Kernels from standard, hand-optimized benchmarks are compared against their counterparts which are compiled by the new compiler using the GTX 480 and 680I. The best possible level 1 cache - shared memory ration has been used. The new compiler achieves an average speedup of 1.76 on the GTX 480 and 1.61 on the GTX 680, significantly outperforming the manually optimized kernels. This shows that the auto-tuning algorithm can optimize memory usage much more efficient than tweaking by hand could, probably because it is very difficult for programmers to accurately assess factors like register pressure. A second experiment was done to see what the impact was of optimization 1 and 2 on the one hand, and optimization 3 on the other. All tested benchmark kernels get better performance from the first 2 optimizations since these are very generally applicable. The third optimization can not be applied for all kernels but achieves higher speedups than the first two since potential global memory access are prevented, which are more costly than on-chip memory accesses.

One of the limitations of this compiler is that only 3 of the possible 6 on-chip data movements are considered. Furthermore, although the approach is performance portable for current GPU generations, if shared memory size would become bigger than the kernel file many of the optimizations will not work anymore. However these missing data movement possibilities have been marked as future work by the authors, recognising this problem.

### *B. Complexity Analysis and Algorithm Design for Reorganizing Data to Minimize Non-Coalesced Memory Accesses on GPU [4]*

One of the features that GPUs offer to increase effective memory bandwidth is memory coalescing. Global memory consists of segments, consecutive memory regions. If the data requested by a kernel resides on as few segments as possible, the memory access is coalesced. Not coalescing memory accesses can be detrimental for performance since it results in loading more segments than needed, lowering the effective memory bandwidth. Memory coalescing problems come in two forms: Static and dynamic. Static problems can be optimized at compile time and have known access patterns. Dynamic problems can not be optimized at compile since the access pattern depends on variables, most often user input. These kernels can only be optimized when this variable is known, which is when the program is already running. Wu. et al. [4] have created a runtime system consisting of a selector, which analyses the kernel, and an assembly of optimization algorithms from which the selector can choose.

The most obvious technique for coalescing memory accesses is to reorder data at runtime so that data accessed by a warp resides in consecutive memory space. However this problem can get very complex if multiple warps access the same data, since there would not be one optimal pattern then. It is even proved that this data reorganising problem is NP-complete, making it unfeasible for runtime systems. This is the case even if this technique is combined with warp reorganization.

However this problem can be circumvented, using a trade-off between space, time and complexity. The main observation here is that traditional reordering algorithms guarantee that the new reordered data does not take up more space than the original data. When this property is loosened, the NP-complete property can be circumvented. First, one already existing algorithm will be covered, then two newly created ones. The first one is created by Zhang et al. [19] which was previously mentioned in related papers section. They use a duplication algorithm which guarantees coalesced memory accesses but has a space complexity equal to the number of threads. Even worse, this number gets multiplied if there are multiple references to the same object. The first new algorithm tries to optimize the duplication algorithm, reducing the space complexity while keeping the same level of optimization. It lets go of the one-to-one mapping between threads and data elements since if two threads in a warp access the same data object, this data object does not have to be duplicated for each thread to get a coalesced memory access. However this does not occur often since there are only 32 threads in a warp, and it may cause unaligned memory access. The first problem is solved by sorting data elements based on the number of threads that access them, followed by reordering the threads according to their accesses to these sorted elements. The second problem is solved by moving data accessed by a warp to the next segment if it does not fit in the current segment, adding padding there instead. Data is only duplicated if it is accessed by two adjacent warps which do not share the same segment. The second new algorithm uses the on-chip shared memory to avoid duplication per thread block because shared memory is private per thread block. It copies the data from global to shared memory, which reduces the access latency later on enormously since shared memory is on-chip.

The selector will choose one of these three algorithms based on information about the application and hardware. The duplication algorithm is used if the number of threads is about the same as the size of the irregularly accessed data since this would circumvent the bad space complexity of this algorithm. Else the padding algorithm is chosen if the kernel has a single access pattern, since reorganizing the threads may break other access patterns which use a thread's id to access data, since only the access pattern that is being optimized will have this fixed automatically. Else the sharing algorithm will be chosen if the irregularly accessed data fits on shared memory, since not having this would result in loading data from global to shared memory, nullifying the optimizations of this algorithm. If all these criteria are not met, duplication is used by default.

For experiments nine kernels with dynamic memory accesses are used with and without the runtime system on a GTX 480 and Tesla C1060. The achieved speedup is up to 21 percent on the former and up to 109 percent on the latter because it lacks on chip level 1 cache. Most other experiments use this machine, which unfortunate because most future mainstream GPUs would have level 1 cache. A speedup is achieved for all kernels: Duplication wins three times, Padding one time and sharing 5 times. The duplication algorithm tends to win when there is no space pressure in the original kernel, while the sharing algorithm tends to win when there is such pressure since it eliminates all duplication per thread block. This is also the case for the padding algorithm but due to its constraint of only having one access pattern in a kernel, it can only be applied to one of the kernels, where it is the fastest.

### *C. GLES: A Practical GPGPU Optimizing Compiler Using Data Sharing and Thread Coarsening [5]*

A result from warps on GPUs executing in lockstep is that diverging control flows become a big problem because multiple branches can not be executed in parallel. This often results in multiple branches being executed and later on thrashed since there is no advanced branch predictor present like on a CPU. This is a handicap for optimization techniques since very often the programming model of a GPU relies on continuous threads executing the same instructions at the same time. Lin et al. [5] have created the GLES compiler which uses data sharing for data reuse and bandwidth enhancement and granularity coarsening for reducing redundant instructions. This is preceded by a divergence analysis which ensures the correctness of these transformations when diverging control flows are present, something which was missing in previous compilers, often resulting in incorrect optimized kernels. GLES is a source-to-source compiler which assumes that the input kernel does not have synchronization barriers and does not use shared memory, while the thread granularity should already be optimized for the current state of the kernel.

Divergence can only occur in statements which are dependent on divergent data, i.e. data which can have different values at the same moment during execution. For divergence analysis this means finding such a case in the data dependency graph of a kernel. When the divergent control flows have bee found, the compiler starts with the data sharing optimization.

With data sharing, data is moved from global to shared memory to reduce data duplication and access latency. Moving the data relies on multiple continuous threads, each moving a part of the whole dataset. When diverging control flows occur, this can not be guaranteed any longer since some threads may take a branch which does not use the same shared memory logic. Furthermore, using shared memory should be accompanied by synchronizing steps to ensure all data has been moved to shared memory. However if not all threads execute this branch, and so not all threads have these synchronizing statements, a deadlock will occur. Since GLES can find the divergent control flow, it will work around these problems if possible, else it will not do these optimizations since correctness should always be guaranteed.

For thread coarsening the same problem of possible unsafe optimizations because of diverging control flow applies, so first the divergence information is updated after the data sharing optimizations. GLES achieves thread coarsening by using instruction sharing. This is achieved by sharing identical statements between threads as much as possible while preserving statements which can be executed in multiple ways. The coarsening will be done on the dimension that contains the least amount of non-shareable statements, which can be optimized to a more precise cost model in future work. Coarsening the thread granularity results in doubling the kernel granularity, which increases register pressure. GLES iteratively coarsens the kernel and calls the NVCC compiler to check if the register pressure is not too high yet. This is done until the register pressure is too high, on which point the previous iteration of the kernel will be used.

For the experiments five heavily optimized kernels were reverse engineered to their naive form and then optimized by the compiler. The experiments were executed on a GTX 480 and 680I. On average GLES is as fast as the manually optimized kernels, achieving about the same speedup for each kernel. Moreover thread coarsening performs better than data sharing, especially on the GTX 680 since its register file is doubled compare to the GTX 480. A second experiment was done to compare GLES with Gcompiler, a state-of-art compiler at the time of publishing which focuses on the same optimizations as GLES. GLES achieves a higher or close performance while the kernels are very simple and do not contain divergence, on which Gcompiler would most likely produce incorrect code.

### *D. PORPLE: An Extensible Optimizer for Portable Data Placement on GPU [6]*

Chen et al. [6] present PORPLE: A data placement optimization engine which is architecture- and input-adaptive and has generalizable optimisations. It consists of a mini specification language, a source-to-source compiler and a runtime data placer. The specification language is used to describe the details of the memory system, making this engine performance portable by countering the problem of significant hardware differences between GPU architecture generations. Input-adaptability is gained by using a runtime data placer, since dynamic memory accesses can not be optimized by a compiler. However for the runtime system not to generate so much overhead that possible execution time optimizations get nullified, a compiler is used to prepare the kernel for the runtime analysis. This combination of a compiler and runtime system also helps in achieving generality by optimizing both static and dynamic memory access problems.

The offline part of the engine consists of the memory specification language (MSL) and the compiler (PORLPLE-C). The advantage of MSL is it can capture all existing GPU memory architectures and their variations, and since only one specification is needed per GPU type, one should use a specification created by a hardware architect and not make one themselves. The MSL configuration will be used by the runtime system (PLACER) to make trade-offs between different data placement schemes.

The compiler is used to lessen the workload of the runtime system. One of the properties of PORPLE is that it is placement-agnostic, which means that the runtime system can choose from multiple data placement options which are all correct. This is achieved by letting the compiler generate all these options beforehand as separate kernels, then at runtime one of these kernels can be picked. This will be preceded by an analysis to guarantee that all generated kernels are correct. Furthermore, the access patterns of the kernel are analyzed by the compiler which will later on be used by the runtime system to assess all data placement options.

PLACER receives a memory specification in MSL, the staged program which contains all data placement options and static access patterns. With this information PLACER will assess the quality of the data placement plans and find the best one. First, a lightweight performance model will be used which is based on the reuse distance model. It takes a data placement plan and the kernel's access pattern to generate the number of access per memory type. This will then be converted to a memory throughput factor using a custom formula which also takes overlap between computation and communication into account with a concurrency factor. However this factor must be determined by the user, which can make the formula less accurate. Now that all data placement plans have a score, any search algorithm can be used to quickly enumerate over the plans since PORPLE supports custom search algorithms. The runtime system also does on-line profiling on the CPU to find out dynamic array sizes and access patterns. For the latter a profiling function is needed, which the compiler tries to generate if it sees a dynamic access pattern. If the compiler fails in generating such a profiler, it will ask the user to provide one. With this PORPLE offers generalizable optimizations since both static and dynamic behaviour is captured. The disadvantage of the dynamic analysis is that it costs time to run the profiler, reducing the potential speedup. The overhead is reduced as much as possible by only simulating the first thread block and only executing the first ten iterations of a loop body if present.

PORPLE will be evaluated against a rule based memory selector algorithm using ten regular benchmarks and 6 irregular benchmarks on a Nvidia Tesla K20c. Only the most important kernel in each application will be optimized since PORPLE can not optimize between kernels. It will also be compared against the theoretical optimal solution which has been calculated using offline exhaustive search. For the regular benchmarks PORPLE achieves an average speedup of 13 percent, almost matching the theoretical upper limit (14 percent) and winning from the rule based method (5 percent). The runtime overhead is very small here (only 1 percent of the runtime) since no dynamic analysis had to be performed. For the irregular benchmarks the difference between PORPLE and the rule based is much bigger (59 and 33 percent speedup respectively) because rule based methods can not capture the complexity of dynamic memory accesses completely. The difference between PORPLE and the theoretical upper limit is also bigger (9 percent) because of PORPLE's runtime analysis limitations due to overhead. This shows exactly what the strong and weak points of PORPLE are: Although the dynamic memory access analysis and the performance model are limited in scope and accuracy, improving them would increase the runtime overhead, which would then counteract the benefit of improving these runtime parts.

### *E. Coherence-Free Multiview: Enabling Reference-Discerning Data Placement on GPU [7]*

Most compiler and runtime systems can only optimize one access pattern per data object, reducing the performance of possible other access patterns on the same object. This limits optimization efficiency of these kernels. Chen et al. [7] present coherence-free multiview, a technique which allows multiple views of a data object to co-exist in memory without coherence guarantees among those views. This enables reference-discerning data placement, with which multiple access patterns of the same data object can be optimized. The big disadvantage of this technique is the reduced coherence guarantees between the different views, e.g. algorithms which do rely on writing and immediately reading the same data on another thread can not be correctly optimized.

A data object has a coherence-free multiview if it has multiple views in the memory systems (access patterns) which are all actively used but do not have guaranteed coherency. This allows a data object, which is read from and written to, to reside in global and constant memory at the same time for instance, which is not possible without coherence-free multiview since constant memory is read-only. The problem of multiple objects referring to the same data is also automatically solved this way. The following conditions apply for safely transforming a kernel to coherence free multiview: There are no intra-warp true dependence (read-after-write) on the object and if there are inter-warp true dependences on the object, there are no synchronizations dictating the order between reads and writes to that object. This holds for read-only, write-only, read-before-write and only some cases of read-after-write objects, however for write-only objects having multiple views has no advantage so these will be ignored.

The implementation extends the PORPLE-C [6] compiler in two ways to support coherence-free multiview. First, an applicability check is implemented which uses the aforementioned conditions. This consists of first removing write-only objects, then checking intra-warp and finally inter-warp dependencies using a dependence analysis. Second, the static data access pattern analysis from PORPLE-C is changed to operate on reference level instead of object level, which will be used in the runtime system to guide data placement decisions. The runtime system of PORPLE, called PLACER, consists of two parts: assessing the quality of a data placement plan and search for the best plan. The problem with the default model which is used to assess the quality of the plans is that it generates too much overhead if many arrays are present in the kernel, which is often the case for coherence-free multiview since one object may have been duplicated for some views. That is why a spec-guided pattern group is made beforehand which categorizes each instruction as a read-zero (constant), read-regular, read-irregular, write-regular or write-irregular. This reduces the complexity of the performance model such that overhead will not be a problem anymore. The algorithm which searches for the best placement plan is also modified to take a copy overhead into account, since one object may need to be duplicated in memory for some views.

Coherence-free multiview will be compared against PORPLE and a rule-based approach using kernels with regular and irregular memory accesses which have slack in their coherence requirements, kernels with slack as a result of optimizations and without slack at all. Nvidia's Tesla M2075, K20c and GTX980 will be used, which all have different memory systems. For all kernels the rule based approach is outperformed by both PORPLE and Coherence-free multiview. For the kernels without slack in their coherence requirements, the performance of multiview is the same as with PORPLE since the multiview optimizations were deemed incorrect by the compiler. For the kernels with slack, multiview systematically outperforms PORPLE, getting an average speedup of 1.49 on the GTX980 compared to PORPLE's 1.27. The overhead in both time and space does not pose a problem. The performance across the GPU's because of different memory system, however multiview takes advantage of PORPLE's inherent performance portability, so the differences in performance are not bigger than they should be.

### *F. Data Placement Optimization in GPU Memory Hierarchy using Predictive Modeling [8]*

The traditional way to assess data placement plans at compile- or runtime is to use a simple model which counts the number of data accesses per memory type to get a memory access cost for a kernel. Stoltzfus et al. [8] use a machine learning approach to create a classifier which can predict which type of memory reduces the execution time of a kernel the most. The approach is split in two parts: Training the classifier once offline and using it in a runtime system to do the analysis and data placement on an input kernel.

The classifier is trained offline once using a selection of kernels which have varying data placement schemes, input data and thread block sizes. All these combinations are tested first to get their execution times. For the global memory variant features are extracted using the nvprof and CUPTI API tools and microbenchmarks. This feature vector is then reduced to 5 categories, which include the number of active warps, executed local, global, shared and texture memory load and store instructions, percentage of stalls because of missing instruction input, percentage of active warps per cycle and the average number of global memory load transactions per memory load. This is done to reduce the complexity of the classifier so it can be used at runtime. The feature vectors are then labeled using the best performing kernel variant. Finally the classifier is trained using these feature vectors and label with multiple algorithms including Random Forest, Support Vector Machines, Attributed Classifiers and some more. The most accurate algorithm is used for the classifier and included in the runtime system.

When a program is executed, the same CUPTI API tool is used to profile the kernel and extract the 5 categories used in the model. The kernel's data is placed in the memory type predicted to be the fastest by the classifier. The evaluation is done on Nvidia's Kepler, Maxwell, Pascal and Volta CPU's, being more recent than the architectures used in previous core papers. For each selected kernel multiple variants are created which use different types of memory if possible. These are executed on the GPU's to get the best performing memory variant per kernel per GPU. The results are compared against the prediction of the classifier which uses the default version of each kernel. The classifier predicts correctly in 80 to 95 percent of the cases per memory type, scoring lower for shared and constant memory kernels than for global and texture memory versions. The classifier has the same accuracy on a benchmark set which was not used to train the classifier.

However this machine learning approach is not yet implemented into a compiler or runtime engine based system, so it is not known if the overhead of the classifier at runtime nullifies possible data placement optimization gains. If this is the case, the classifier has to be made more lightweight which would most likely lower its accuracy. In conclusion, it is not yet possible to compare this machine learning model to other compiler and runtime system methods but it has proven that machine learning has a potential for very accurate data placement predictions.

## V. A COMPARATIVE ANALYSIS

We have analysed in detail six papers about compiler and runtime system optimizations for GPU data placement. Our next step is to compare these papers. Table II presents the core papers using their reference number[2]. Further, we have chosen eight criteria to compare the papers by: usage of compiler or runtime system, architecture adaptiveness, usage of an adaptive framework and presence of register file, shared memory, texture memory, multiple access pattern or divergence optimizations.

The most important difference between the core papers is the usage of a compiler, a runtime system, or both since this determines if static or dynamic data placement problems can be tackled. For example, in [3] and [5], only a compiler is

[2]For the remainder of this comparative analysis, we only use the reference numbers, to simplify the text.

TABLE II
A COMPARISON OF CORE PAPER CHARACTERISTICS IN ORDER OF OCCURRENCE.

| Paper | Compiler or runtime | Architecture adaptive | Framework | Register file | Shared memory | Texture memory | Multiple access patterns | Divergence |
|---|---|---|---|---|---|---|---|---|
| [3] | Compiler | Yes | No | Yes | Yes | No | No | No |
| [4] | Runtime | No | No | No | Yes | No | Yes | No |
| [5] | Compiler | No | No | Yes | Yes | Yes | No | Yes |
| [6] | Both | Yes | Yes | No | Yes | Yes | No | No |
| [7] | Both | Yes | Yes | Yes | Yes | Yes | Yes | No |
| [8] | Runtime | Yes | No | No | Yes | Yes | No | No |

used, limiting the access patterns they can analyse, and thus limiting the applicability of their optimizations. Naturally, their optimizations can have full effect on kernels which do not use dynamic access patterns; but for kernels which do, the optimizations are less effective or even not applicable. On the other hand, in [6], a framework is used which consists of both a compiler and a runtime system, which is also used in [7]. This approach combines the best of both worlds: being able to optimize both static and dynamic problems while keeping the runtime overhead low by letting the compiler do as much work as possible beforehand. Runtime overhead is an important aspect for these papers: using a very complex model to assess the quality of data placement plans at runtime would increase performance by predicting the best possible data placement plan more accurately, but it would also increase overhead since a more complex model takes more time to execute. This is also the conclusion of [8], where even a classifier with 5 features is too complex for runtime usage.

Another important aspect of these systems is their ability to adapt to different GPU architectures, which change very rapidly. The best example can be found in [3]: although the algorithm adapts to the architecture by using auto-tuning, one of the assumptions of the algorithm is that moving data from the register file to the level 1 cache or shared memory is not needed since the register file is bigger than the other two combined and shared memory is already overused in most kernels. However, if in the future the architecture changes such that the register file becomes much smaller than the level 1 cache and shared memory, the algorithm becomes far less useful since it ignores a very important optimization option. In some cases architecture adaptiveness plays a less significant role, such as for [5] since divergence is a core part of GPU design and will most likely not be changed in the near future. The most adaptive system has been built by [6] since it is placement agnostic so as long as the model used to assess the placement plans is kept up to date, which is easy to do since it can be user defined, the optimizations keep working. However this comes at the cost of providing a detailed architecture specification document.

In Table II five specific optimizations have been highlighted: for the register file, shared memory, texture memory, multiple access patterns and divergent behaviour. Shared memory optimizations have been used in all core papers since this is arguably the most effective addition to the memory subsystem on GPUs compared to CPUs since it is scratchpad memory which can be used to share data across threads in a thread block, resulting in enormous speedups compared to using level 1 cache for appropriate kernels. Furthermore, shared memory was already part of GPU architectures prior to Fermi, while the level 1 cache was not, meaning that using if effectively was critical for getting good performance. Optimizations for the register file have also been used in some papers because of the interesting design it has compared to its CPU counterpart. These optimizations are all done in some auto-tuning fashion because of the nature of register file optimizations. The final highlighted memory type is the texture memory which is used for read-only data. Very often read-only data is stored in global memory since the user has to use specific flags on a data object for it to be moved to the texture memory. A few of the papers recognize this problem and do an extra read-only data check to get data to texture memory, which not only has the advantage that texture memory is more suited for dealing with that kind of data but in some architectures it has its own data pipe to streaming multiprocessors, potentially doubling the bandwidth from off-chip to on-chip resources.

Both [4] and [7] have as a goal to optimize kernels with multiple access patterns on the same data object. Optimizing this is very difficult when keeping the data size constant so both papers relax this property, however their approach is very different. While [4] try to minimize the space overhead as much as possible, using only 1 transformed copy of the original data, [7] use multiple copies of the same data, not having any coherence guarantees between the copies. Another specialized optimization has been implemented in [5], namely data and instruction sharing with correctness guarantees in kernels with divergent instruction paths.

These are all very specific optimizations: Only by combining them into one framework a competitive and usable option for users can be created. Currently if someone wants to further optimize a kernel without tweaking it by hand, he or she would need to investigate which algorithm from which paper can provide a good speedup for this specific kernel. Such a time consuming step can be prevented by combining lots of optimizations into one framework which would provide more speedup for more different applications, resulting in a higher chance of actually being used by someone because of increased usability. The best possible framework on which to build on is provided by [6]: The system is architecture and input adaptive, supports static and dynamic optimizations and some key algorithms can easily be changed via user defined functions. [7] already build upon this framework by adding its own optimization to the already provided ones. Not all optimizations can be combined of course, however adding algorithms like divergence analysis [5] can be a very valuable addition.

These are all comparisons between papers in the specific category of compiler and runtime system optimizations, but what about general trends in GPU data placement research? Figure 2 highlights the number of published papers per year on GPU data placement that can be found using Google Scholar. The tool [23] was used to gather these results. While this is not completely accurate since Google Scholar is not the only source of papers and not all papers which match this query are automatically about this topic, the results are accurate enough for extracting trends. Apparently the number of published papers on this topic has decreased over the years, while the number of architecture updates has increased. On first glance this seems contradictory since there being more architecture updates would make one think that more research is done on the creation and validation of those new designs. A possible explanation for this contradiction could be that the hardware is evolving so fast that research can not keep up with it. There is a lot of time in between the start of a research and the publication of a paper, so a new technique could already be obsolete before it is published. No research group wants to take this risk, which could explain the downward trend. Furthermore, GPU manufacturers do not disclose much information about hardware implementations, meaning that those details need to be reverse engineered before new low level research can be started. This too takes much time so it is not worth it if new architectures are created almost every year. The rise of machine learning applications on GPUs play a role as well: Most of these applications use high-level libraries like TensorFlow to handle the low-level implementation for them, so the application developers themselves do not have to do much research into low level GPU data placement anymore. Previously this was not the case since no single domain dominated the GPU application space as much as machine learning currently does.

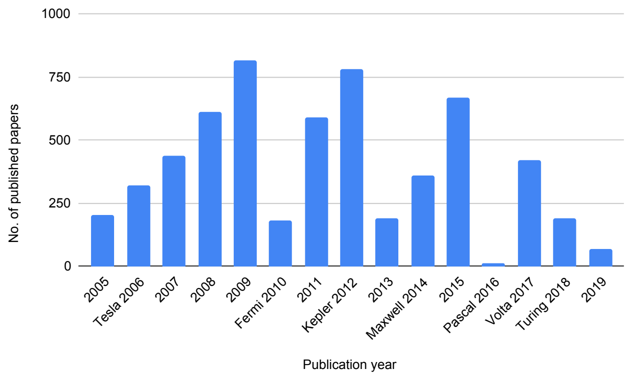

Fig. 2. Number of published papers per year for query ”GPU data placement” on Google Scholar.

## VI. Conclusion

This literature study presents an overview of research done on the topic of GPU data placement to answer to question if the memory gap can be bridged by using techniques to fully utilize the available memory hierarchy of a GPU. Some of the most important categories of papers on this topic include compiler and runtime systems, thread granularity and thread level parallelism, hardware modifications and performance modeling. The first category has been analyzed in-depth by selecting six papers which all use different compiler and runtime techniques to optimize data placement of a kernel. This category has gotten the most attention because creating software solutions is much easier than building hardware ones, and these techniques have already proven themselves to be effective on CPUs. The analyzed optimizations can be categorized as follows: register file, shared memory, texture memory, multiple access patterns, and divergence. Furthermore, the difference between compiler and runtime system optimizations must be taken into account since it determines the applicability and overhead of the optimizations. The proposed optimizations have shown potential, achieving high average speedups compared to hand-optimized kernels per paper. However, only [6] have put effort into the creation of a framework to which other researchers can add functionalities, which has already been done by [7]. Only when the current mostly self-contained optimizations get combined into such a framework the memory gap can be attempted to be bridged for a large variety of kernels. One of the reasons for the lack of such a framework is the high complexity of the GPU memory subsystem and the speed at which it has changed throughout different architectures. Although the optimizations and frameworks try to be architecture adaptive, adaption is time consuming and often impossible because of rigorous architecture changes. To overcome this problem two possible solutions have been identified: either increase the amount of research done on this topic to create better architecture adaptive solutions or implement hardware solutions for data placement as was done on the CPU. The current problem with the former is that the amount of research done has actually decreased over the years while the number of architecture updates have increased, while the latter is difficult to achieve since details about GPU hardware implementations are not disclosed by the manufacturers.

## References

[1] M. V. Wilkes, “The memory gap and the future of high performance memories,” *ACM SIGARCH Computer Architecture News*, vol. 29, no. 1, pp. 2–7, 2001.

[2] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, “Fermi gf100 gpu architecture,” *IEEE Micro*, vol. 31, no. 2, pp. 50–59, 2011.

[3] C. Li, Y. Yang, Z. Lin, and H. Zhou, “Automatic data placement into gpu on-chip memory resources,” in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 23–33, IEEE Computer Society, 2015.

[4] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen, “Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu,” *SIGPLAN Not.*, vol. 48, pp. 57–68, Feb. 2013.

[5] Z. Lin, X. Gao, H. Wan, and B. Jiang, "Gles: A practical gpgpu optimizing compiler using data sharing and thread coarsening," in *International Workshop on Languages and Compilers for Parallel Computing*, pp. 36–50, Springer, 2014.

[6] G. Chen, B. Wu, D. Li, and X. Shen, "Porple: An extensible optimizer for portable data placement on gpu," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 88–100, IEEE Computer Society, 2014.

[7] G. Chen and X. Shen, "Coherence-free multiview: Enabling reference-discerning data placement on gpu," in *Proceedings of the 2016 International Conference on Supercomputing*, p. 14, ACM, 2016.

[8] L. Stoltzfus, M. Emani, P.-H. Lin, and C. Liao, "Data placement optimization in gpu memory hierarchy using predictive modeling," in *Proceedings of the Workshop on Memory Centric High Performance Computing*, pp. 45–49, ACM, 2018.

[9] A. B. Hayes and E. Z. Zhang, "Unified on-chip memory allocation for simt architecture," in *Proceedings of the 28th ACM international conference on Supercomputing*, pp. 293–302, ACM, 2014.

[10] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, "Unifying primary cache, scratch, and register file memories in a throughput processor," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 96–106, IEEE, 2012.

[11] C. Li, Y. Yang, H. Dai, S. Yan, F. Mueller, and H. Zhou, "Understanding the tradeoffs between software-managed vs. hardware-managed caches in gpus," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 231–242, IEEE, 2014.

[12] S. Unkule, C. Shaltz, and A. Qasem, "Automatic restructuring of gpu kernels for exploiting inter-thread data locality," in *International Conference on Compiler Construction*, pp. 21–40, Springer, 2012.

[13] Y. Yang and H. Zhou, "Cuda-np: realizing nested thread-level parallelism in gpgpu applications," *ACM SIGPLAN Notices*, vol. 49, no. 8, pp. 93–106, 2014.

[14] Y. Huang and D. Li, "Performance modeling for optimal data placement on gpu with heterogeneous memory systems," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 166–177, IEEE, 2017.

[15] J. Lai and A. Seznec, "Performance upper bound analysis and optimization of sgemm on fermi and kepler gpus," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 1–10, IEEE, 2013.

[16] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for gpgpus," in *Proceedings of the 22nd annual international conference on Supercomputing*, pp. 225–234, ACM, 2008.

[17] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A gpgpu compiler for memory optimization and parallelism management," in *ACM Sigplan Notices*, vol. 45, pp. 86–97, ACM, 2010.

[18] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 105–118, 2010.

[19] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for gpu computing," in *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 369–380, ACM, 2011.

[20] Y. Yang, P. Xiang, M. Mantor, N. Rubin, and H. Zhou, "Shared memory multiplexing: a novel way to improve gpgpu throughput," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 283–292, IEEE, 2012.

[21] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and J. S. Vetter, "Exploring hybrid memory for gpu energy efficiency through software-hardware co-design," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pp. 93–102, IEEE Press, 2013.

[22] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 72–83, IEEE Computer Society, 2012.

[23] V. Strobel, "Pold87/academic-keyword-occurrence," 2018.