DOI: 10.1111/cgf.14671
Pacific Graphics 2022
N. Umetani, E. Vouga, and C. Wojtan
(Guest Editors)

*Volume 41 (2022), Number 7*

# Fine-Grained Memory Profiling of GPGPU Kernels

Max von Buelow[1], Stefan Guthe[1] and Dieter W. Fellner[1,2]

[1]Technical University of Darmstadt, Germany
[2]Fraunhofer IGD, Germany & Graz University of Technology, Institute of Computer Graphics and Knowledge Visualization, Austria

**Abstract**

*Memory performance is a crucial bottleneck in many GPGPU applications, making optimizations for hardware and software mandatory. While hardware vendors already use highly efficient caching architectures, software engineers usually have to organize their data accordingly in order to efficiently make use of these, requiring deep knowledge of the actual hardware. In this paper we present a novel technique for fine-grained memory profiling that simulates the whole pipeline of memory flow and finally accumulates profiling values in a way that the user retains information about the potential region in the GPU program by showing these values separately for each allocation. Our memory simulator turns out to outperform state-of-the-art memory models of NVIDIA architectures by a magnitude of 2.4 for the L1 cache and 1.3 for the L2 cache, in terms of accuracy. Additionally, we find our technique of fine grained memory profiling a useful tool for memory optimizations, which we successfully show in case of ray tracing and machine learning applications.*

**CCS Concepts**

• ***Hardware*** → ***Simulation and emulation;*** • ***Computing methodologies*** → ***Graphics processors;*** • ***Theory of computation*** → *Program analysis;*

## 1. Introduction

Standard general purpose GPU profilers (like NVIDIA Nsight Compute) commonly evaluate hardware performance counters that are available on the granularity of a kernel call in order to display cache hit rates, the number of memory transactions and other memory-related metrics to the user. Supported by these metrics, software engineers usually aim to enhance inefficient parts of their software programs. While this is a standard way of investigating potential bottlenecks in GPU applications, the exact cause of an inefficiency remains invisible to the software engineer. Deep knowledge into hardware and the program itself is required in order to find and optimize these inefficiencies. In many cases it would thus be useful to have a finer grained representation of memory-related profiling metrics that records memory metrics for each memory allocation separately, allowing the user to narrow down the problem into specific regions of the program where this allocation is actually used.

In this paper, we introduce *profiner*, a profiling pipeline that automatically derives such fine-grained memory profiles from compiled GPU applications during run-time—analogous to existing profilers. For this purpose, our profiler extracts a list of memory references from the target application and simulate the whole memory flow from the processor through the caches to the DRAM. We rely on realistic hardware parameters from NVIDIA GPU architectures that have been extracted using previous micro benchmarks and model them in our simulator.

In summary, our contributions are:

- A profiling tool that implements allocation-wise display of cache hit rates and coalescing behavior.
- A detailed model of the memory of recent GPU architectures that outperforms state-of-the-art memory models in terms of accuracy compared to actually measured profiling metrics.

## 2. Related Work

**Memory Modeling** Performance modelling is a well-researched topic. While simulators like GPGPU-Sim [BYF*09] and Accel-Sim [KSAR20] aim to model the whole GPU, other works—including ours—specifically focus on accurate memory models. In order to model memory given a list of memory transactions, many cache simulators rely on reuse distances in order to simulate the behavior of LRU caches. This technique is popular for CPU architectures [DZ03], as well as for GPUs [NvdBCB14; TYL11; WX16]. In particular, the model of ARAFA, CHENNUPATI, BARAI, et al. [ACB*19] samples estimations from the Stack Distance Cache Model (SDCM) of AGARWAL, HENNESSY, and HOROWITZ [AHH89], which models associativity of LRU caches, on basis of a reuse distance histogram. Based on this, PPT-GPU-Mem [ABC*20] uses the *NVIDIA Binary Instrumentation Tool* (NVBIT) [VSNK19] in order to simplify the memory trace extraction process.

**Benchmarks and Ray Tracing** While existing GPGPU benchmark suites [GXS*12; CBM*09; CBRS13; KKS*19] already cover a wide range of individual GPU applications, our fine-grained memory profiler was specifically, but not exclusively, designed for GPU-based ray tracers. In contrast to these benchmarks, ray tracers usually produce very heterogeneous work loads that access a wide range of different allocations in the same kernel call, ranging from geometry to hierarchical acceleration structures. Additionally, memory and especially the amount of data movement from DRAM and caches is the primary bottleneck of GPU-based ray tracers [VSM*18; AL09]. This makes ray tracers an interesting target for further fine-grained analysis based on a set of different optimizations that we shortly summarize in the following. A popular method to reduce memory traffic in ray tracing is the use of bounding volume hierarchies (BVH) and optimizations of these (surface area heuristic, e.g.) [MB90; AKL13]. Besides the BVH, memory layout also plays a crucial role in performance optimization of GPU ray tracers. While WODNIOK, SCHULZ, WIDMER, and GOESELE [WSWG13] apply low level optimizations onto memory organization, WALD, MORRICAL, and ZELLMANN [WMZ22] employ compression techniques onto the mesh geometry. We use simplified versions of such optimizations to show their impact onto our extracted fine-grained profiling values.

## 3. GPU Preliminaries

In this section, we briefly describe relevant parts of typical NVIDIA GPU architectures that are mandatory for our memory model.

**Units of Execution** Most GPUs consist of multiple layers of parallel execution units. While the program being run on the GPU is named the *kernel*, the set of all threads executing said program is called the *grid*. The *grid* is split into multiple *blocks*. Inside a *block*, all *threads* can cooperate tightly, mirroring the underlying hierarchy. From a hardware perspective, the first layer, named *streaming multiprocessor* (SM), is similar to a multi-core CPU and implements the MIMD model of FLYNN's taxonomy [Fly66]. As already mentioned, a user-defined number of threads form a *block*. This block is then permanently mapped onto a single SM until it terminates, freeing all its resources like registers and shared memory. Multiple *blocks* may execute on a single SM if a *block* contains too few threads for full occupancy. Each *block* is split into multiple *warps* consisting of up to 32 *threads* per *warp*, operating in SIMD fashion. Each thread within a warp is permanently assigned to a single *lane*.

**Memory** On NVIDIA GPUs, a thread may reference 1 B, 2 B, 4 B or 8 B of aligned virtual memory with a single instruction [NVI22a]. However, due to the SIMD architecture of warps, *memory requests* are executed warp-wise and therefore contain addresses from 32 threads. These addresses are coalesced automatically if they form an aligned sequence. If not, memory accesses must be executed sequentially. The size of a sequence (i.e. a *memory transaction*) depends on the architecture and access type, traditionally up to 128 B. GPUs also involve a cache hierarchy for memory references in order to speed up consecutive accesses. However, details of the cache hierarchy are unpublished and must be explored using fine-grained micro-benchmarking [MC17]. Virtual memory is cached SM-wise in the L1 cache. The L1 cache is a set-associative LRU cache for older architectures and a proprietary implementation of an LRU-like cache for recent architectures. The L1 line size is defined to be 128 B, resulting in referencing only a single cache line if every thread accesses a 32-bit data type (e.g. `float`) in an aligned fashion. The L2 cache is also a set-associative LRU cache and maps physical memory. It is located off-chip and acts between the DRAM and all L1 caches.

**Binary Instrumentation** *Binary instrumentation tools* (NVBIT [VSNK19] on NVIDIA GPUs) replace user-selected instructions with a jump instruction to a *trampoline* program whose purpose is exclusively to save and restore the program state for calling the user-defined *instrumentation function* safely and execute the actually replaced instruction. This way, binary instrumentation can be used to add such function before each memory instruction into the program flow. Prominent use cases for such memory tracing techniques are memory debuggers and profilers.

## 4. Fine-Grained Memory Profiling

This section describes our pipeline of fine grained memory profiling. The following sections coarsely follow our modular pipeline visualized in fig. 1. In section 4.1 we explain how we extract memory traces that we use to simulate the L1 cache (section 4.2) followed by the L2 cache (section 4.3). Finally, we use the information from cache simulation in order to calculate allocation-wise cache hit rates in section 4.4.

### 4.1. Address Extraction

Inspired by PPT-GPU-Mem [ABC*20] (shortly *PPT* in the following), we use NVBIT [VSNK19] for memory reference extraction during running-time of the applications. We instruct the binary instrumentation tool to extract all 32 addresses of threads within a warp including their predicates (marking currently inactive threads), the opcode of the memory instruction used for deriving the number of bytes processed by the memory instruction and processing hardware identifiers (SM and warp) from the GPU kernel. Our profiler simultaneously streams that data to the host system for further processing. We verified extracted memory traces by counting the number of memory requests that have at least one active predicate and comparing it against the profiled number of memory requests of the official Nsight Compute profiler (see section 5.1). In contrast to PPT, we keep the ordering of memory requests from the memory trace, as our profiler is designed to be an on-line tool (i.e. it should be used on the same device the software engineer working on). However, re-scheduling for different GPU architectures can be implemented easily in our modular pipeline by sorting the buffered list of memory accesses.

Accesses to global memory can be directly used for subsequent steps. An exception applies to accesses in the local memory space. On the actual GPU architecture, local memory is strided into 4 B segments such that each 32 bit word is referenced by consecutive threads making coalescing more efficient [NVI22a]. The striding requires splitting up memory accesses greater than 4 B. Additionally, we found out that extracted local memory addresses are offsets within a special virtual memory area that do not differentiate

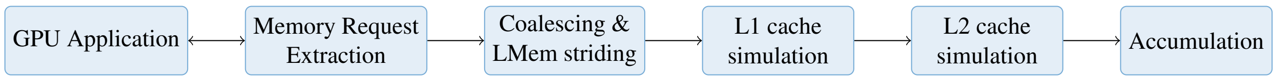


**Figure 1:** *The profiler's modular pipeline. Our profiler is divided into the following five replacable modules. First, the binary instrumentation collects memory references. Then, addresses are pre-processed given their address space, followed by the simulation of both cache levels separately. Finally, our profiler accumulates this data for display.*

between threads such that each thread seems to access the same memory area from a naive perspective. We handled this by subtracting the local memory base address $b$ from each reference $r$ and re-calculating the actual memory address given the number of warps per SM $n_w$, the allocated local memory per thread $L$, the thread index within a warp $i_t$, the warp index within the SM $i_w$ and the SM index $i_s$ as follows.

$$r_{new} = (i_s \cdot n_w + i_w) \cdot 32 \cdot L + (\lfloor r/4 \rfloor \cdot 32 + i_t) \cdot 4 + (r \bmod 4) \quad (1)$$

Our memory extractor then uses said memory request list in order to simulate the memory coalescing behavior of the GPU. We found out that our reference GPU coalesces multiple memory accesses within a warp if they fall within the same 32 bit range; otherwise they are handled sequentially. Similar to the previous step, we reconstructed this coalescing behavior by comparing the number of memory transaction from the official profiler against the total number of coalesced addresses of our simulator (also described in section 5.1).

Given this flat list of memory transactions, our profiler can perform the actual cache simulations.

## 4.2. L1 cache

The L1 cache is very important for cache rate simulations as they directly influence L2 cache rates: Only memory references that miss the L1 cache are forwarded to the L2. Therefore, we aim to use realistic models and parameters of our target architecture. As we aim to model a *NVIDIA Turing* GPU, we consulted the work of JIA, MAGGIONI, SMITH, and SCARPAZZA [JMSS19] which applied micro-benchmarking techniques onto said architecture. Most importantly, they found out that the Turing L1 cache has a non-LRU replacement policy, 32 B lines ($L$) arranged in four sets $S$ and a total capacity $C$ of 57 kB in its default configuration (i.e. no shared memory used). Given these values, we can derive the associativity $A$ (i.e. the number of cache ways, in our case 456) of the L1 cache as follows:

$$A = \frac{C}{L \cdot S} \quad (2)$$

Sets are organized slightly uncommon in the L1 cache. As it evicts four consecutive cache lines at the same time once the cache saturates, we rather call them *sub-lines* and define our cache slightly different compared to a standard set-associative cache. We assume a fully associative cache with 456 128 B lines that can be loaded partially at a 32 B granularity. Our simulator then uses a 4 bit tag per line in order to indicate if these sub-lines are present in the cache or not.

We experimented with multiple cache replacement policies and experienced that reuse distances (i.e. a naive LRU stack) overestimate cache rates in the most cases, which confirms findings that it is not LRU [JMSS19]. We use a tree-based pseudo LRU (PLRU) [KJ10] implementation for the L1 cache, as it is likely to be used in real hardware due to its high lookup efficiency in hardware that can be implemented using cheap bit-wise operations and small additional memory footprint. We additionally found that a PLRU cache results in the most accurate cache rate predictions for our set of benchmark applications when compared against measured profiling values from Nsight Compute, because it results in slightly smaller hit rates than a standard LRU cache. Due to the non-uniform number of nodes (456 lines), the PLRU permutation tree remains non-full. Thus, we were forced to build a left-complete binary tree in breadth first order to generate an appropriate lookup-table for PLRU state transitions. We speculate that the 7 kB discrepancy between the documented 64 kB and measured 57 kB is used for storing the cache state and tags inside the same memory hardware.

## 4.3. L2 cache

Normally, L1 cache misses result in L2 lookups. Despite that, there are also other causes for a L2 lookup. Examples are atomic operations, that always bypass the L1 cache and global memory writes write through the L1 cache therefore always affecting both layers in order to ensure coherence [NVI22b]. Nevertheless, because the local memory does not have to be consistent across SMs as it only stores per-thread data, it uses a write-back mechanism between L1 and L2. Keeping these facts in mind, we use a system analogous to PPT for the L2 cache. The L2 cache of the Turing architecture has a documented capacity of 5.5 MB. Micro benchmarks reveal that the Turing L2 is a 16-way set-associative LRU cache with 64 B lines. We found that a standard LRU cache models matches the actual cache hit rates best. Our profiler calculates reuse distances for each memory transaction independently for each set by allocating the amount of splay trees equal to the number of sets and updating only the one corresponding to the address. If the reuse distance is smaller than the associativity, we assume a hit. This procedure is much more efficient than using the stack distance cache model, because our simulator does not have to compute SDCM's inefficient binomial distributions and splay trees become flatter. Nevertheless, it requires a slight amount of additional memory for storing these multiple trees.

## 4.4. Accumulation

In previous steps, our profiler annotated each memory transaction if it was a hit in the L1 or the L2 cache. Given these transaction-wise cache hit assignments, our profiler is able to perform the desired fine-grained allocation-wise accumulations as follows. Our profiler groups all memory accesses given their corresponding memory allocation and computes the conditional probability $p(hit|i)$ that a memory access is a hit given the allocation $i$ from the number of memory accesses $n_i$ within the allocation and the number of hits $h_i$ as follows.

$$p(hit|i) = {}^{h_i}\!/_{n_i} \tag{3}$$

For comparability against existing profilers, we can then compute the whole-program (coarsely grained) hit rate $p(hit)$ by applying the chain rule as follows. Note, that $n$ is the total number of memory transactions and equal to the sum of $n_i$.

$$p(hit) = \sum_i p(hit|i) \cdot p(i), \quad p(i) = {}^{n_i}\!/_{n} \tag{4}$$

Combining eqs. (3) and (4) shows that the calculations in our memory model are analogous to those in standard profilers which use internal hardware counters in order to account the total number of hits $h$ and divide them by the total number of issued memory transactions $n$.

$$p(hit) = \sum_i \frac{h_i}{n_i} \cdot \frac{n_i}{n} = \sum_i \frac{h_i}{n} = {}^{h}\!/_{n}. \tag{5}$$

As allocations are merely identified by their base address, which varies between kernel calls, we implemented a further optional allocation naming programming interface, that allows, given there is access to the program source code, to bind names to allocated memory regions (i.e. to the base address). Our profiler is then able to automatically show this optional extra information while displaying fine grained profiling values for each allocation. This helps the programmer to assign cache hit rates to allocations and their usage in the source code. However, if allocations cannot be named due to lack of access to the source code, allocations can mostly be identified manually by their size and the number of read and write requests to them.

## 5. Results

In this section, we evaluate our memory profiling simulator against a state-of-the art memory model and actually measured values on a wide range of benchmark applications (section 5.1). Then, we discuss fine-grained cache hit rates on a smaller set of more advanced GPU programs in section 5.2.

### 5.1. Comparison of Coarsely Grained Hit Rates

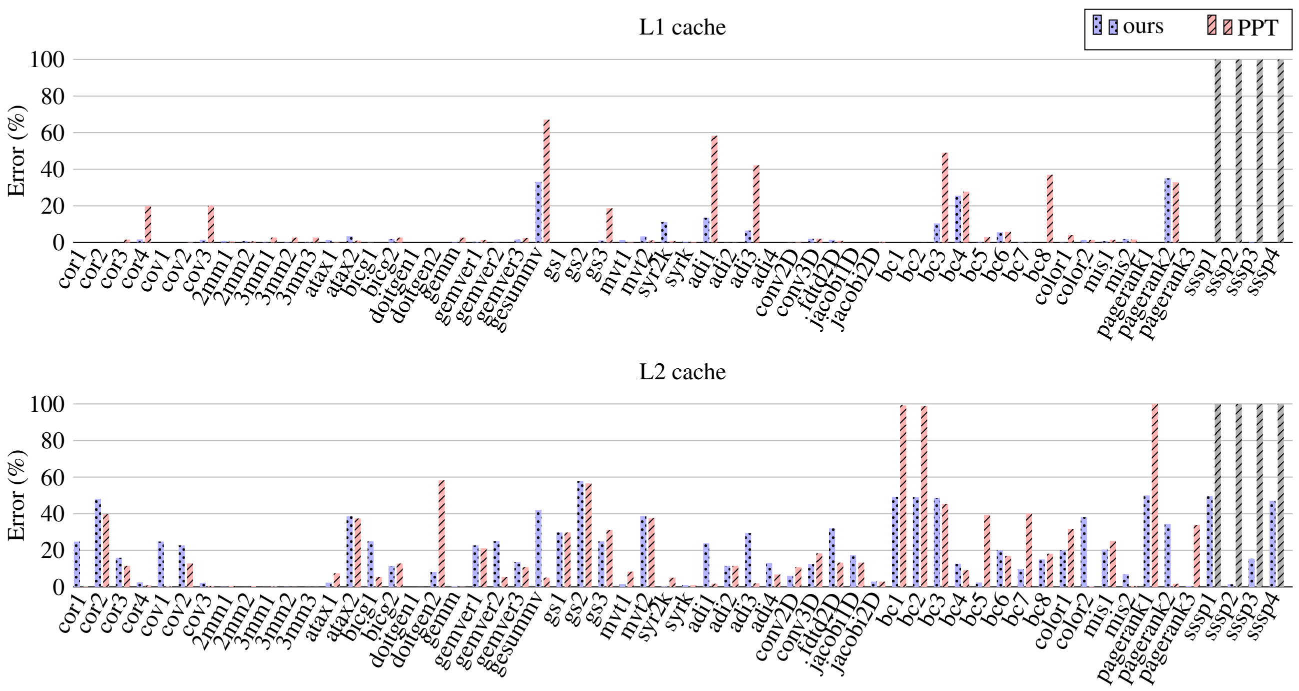
In order to validate the correctness of our memory simulator, we compare cache hit rates from our simulator (see eq. (4)) with those from the PPT model and actually measured ones from NVIDIA Nsight Compute on a NVIDIA RTX 2080 Ti GPU. Therefore, we use a wide range of applications from a set of benchmarks. PolyBench [GXS*12] is a benchmark including numerical algebra applications like matrix and vector multiplications (2mm, 3mm, atax, gemm, gemver, gesummv, mvt, syr2k and syrk) and linear systems solvers (bicg and gramschmidt). It also includes statistics operations (cor and cov) and stencils (adi, convolution2D, convolution3D, fdtd2D, jacobi1D and jacobi2D). The Rodinia benchmark [CBM*09] consists of graph algorithms (b+tree, bfs, pathfinder), applications from physical, biological and medical domains (cfd, hotspot, hotspot3D, needle, heartwall and particlefilter), linear algebra (gaussian and lud), traditional data mining (nn and streamcluster) and compression (dwt2d and huff). Pannotia [CBRS13] focuses on graph algorithms (bc, color, mis, pagerank and sssp). Finally, we also use the AlexNet, LSTM and ResNet neural networks from the Tango [KKS*19] benchmark in our comparison. For a more detailed description on these benchmark applications, we would like to refer to their works directly. We use standard configurations for each (except for cov and cor, as memory traces become extremely large) and, when applicable, the biggest included input dataset. In contrast to the evaluation in the PPT paper, we decided to include this huge and almost complete amount of applications from the benchmarks as we see it impossible to decide which one to leave out. Benchmarks that perform well on our approach are as important for our discussion as benchmarks that perform worse.

Each benchmark application may contain multiple kernels. As we experienced immense heterogeneity on profiling values when executing the same kernel on different data (see the BFS application), we also separate these individual calls in our comparison in cases where it makes a difference. This is also different to the comparison in the PPT paper, as they only include the first call to a kernel in their comparison.

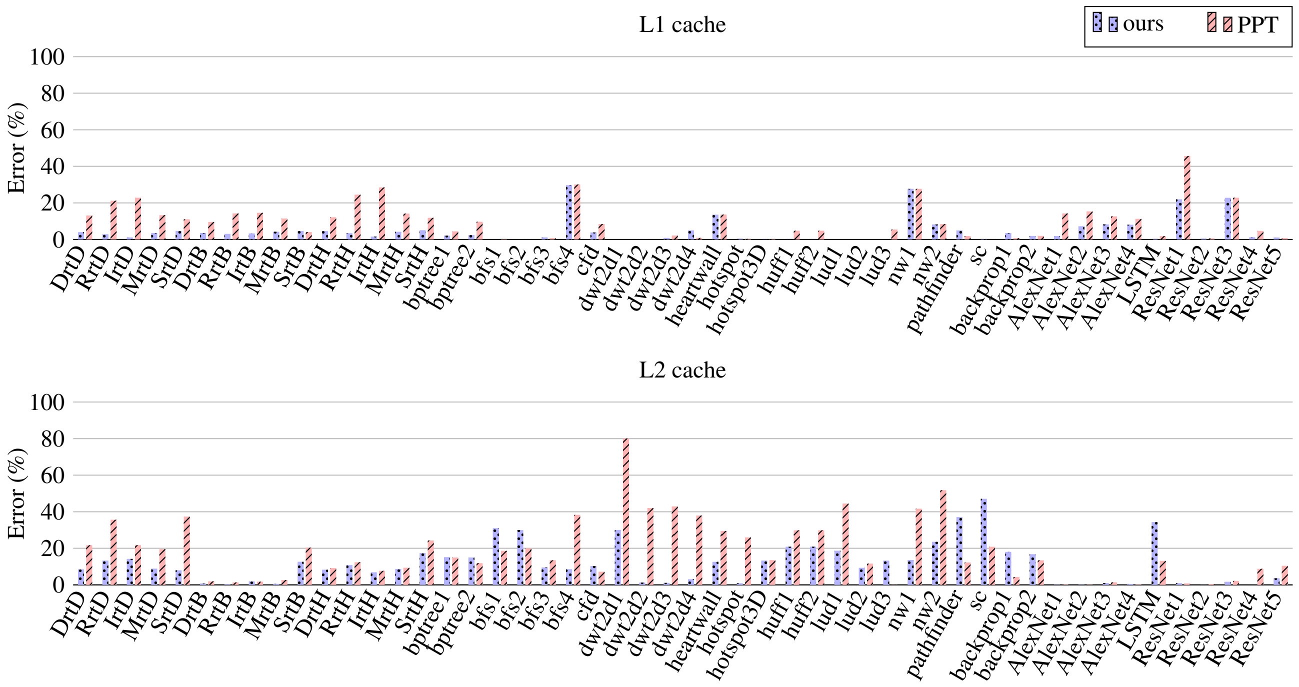
**Memory Requests and Coalescing** First, in order to reduce fundamental sources or inaccuracy, we compared the number of memory requests and transactions with actually measured values. Fortunately, Nsight Compute has options to deliver ground truth values for global and local memory space and read and write operations separately. We found that the number of requests only differ by approximately 0.001 %, which confirms that our recording of memory requests is correct. We assume that this extremely small derivation remains from synchronization barriers that we experienced to issue internal memory requests, that NVBIT does not capture. The number of transactions varies by approximately 0.28 %. We assume that this further small deviation comes from rare side effects in hardware that need be discovered in future work. As deviations appear to be negligibly small, we conclude that the general idea of our coalescing model is correct and use the generated list of transactions for further cache simulation.

**L1 Cache** Figure 2 shows the L1 and L2 cache hit rates of PolyBench, Pannotia (fig. 2a), Rodinia and Tango (fig. 2b). For brevity, we left out all kernel calls where our results, those of PPT and the measured ones are approximately the same or when individual kernel calls have duplicate results as they are less important for our discussion. Raw cache hit rates are included in the supplemental material.

Our L1 comparison clearly shows that our implementation as well as PPT still performs very good on the matrix multiplications (2mm and 3mm). Nevertheless, the diagram also reveals that our L1 cache simulation outperforms PPT in multiple cases (gesummv, gs3, adi, bc, b+tree, euler3d, lud3, alex1, ResNet and our ray tracing implementations). PPT, however, seems to deliver better results for mvt, syr2k, pagerank, backprop. Our implementation outperforms PPT with an mean absolute percentage error (MAPE) of 3.73 % compared to 9.01 % for PPT on our set of benchmark applications.



**(a)** *PolyBench and Pannotia*



**(b)** *Our ray tracer, Rodinia and Tango*

**Figure 2:** *Comparison of the absolute percentage error (APE) between measured cache hit rates and cache hit rates estimated using our approach and PPT-GPU-Mem. Gray bars indicate that the profiler failed to process the target program.*

The execution of PPT failed with a segmentation fault on the sssp benchmark, which we did not further analyze. There are also cases where our implementation and PPT leads to inferior L1 hit rate approximations at the same time, which indicates that our implementation still does not completely reflect the actual memory behaviour.

The table with absolute hit rates values in the supplemental material reveals that PPT over-estimates cache hit rates in many cases. This seems in the first place to be a particularly hard to explain behaviour, as they assume a smaller L1 cache capacity (32 kB) than our implementation (57 kB), probably because they assumed the wrong default of the two available L1 capacity configurations (32 kB or 64 kB). When we analyzed their official implementation, we found that they replaced the stack distance cache model with an approximation they claim to be compute-wise more efficient. Unfortunately, their approximation includes a mistake that drastically over-estimates cache hit rates if stack distances are greater than the cache associativity (i.e. misses). In the end, the incorrect assumption on the L1 capacity compensates the wrongly implemented SDCM approximation. Nevertheless, this mistake does not affect their L2 cache rate simulation as their per-transaction hit decision is based on bare reuse distances. We show a plot of their approximation in our supplemental material and compare it against the actual stack distance cache model. We assume that this implementation mistake makes PPT less robust and less explainable, despite the fact that it still delivers surprisingly good approximations when assuming half the cache capacity.

**L2 Cache** The general trend of L2 cache hit rate approximations–for both PPT and our implementation—seems to be that the inaccuracy increases. This is hardly surprising as the L2 cache relies on misses from the L1 cache and therefore effectively accumulates errors. This fact also becomes clear when looking at the MAPE, which is 15.81 % for our simulator and 20.10 % for PPT. In more detail, our simulator performs better for doitgen, conv2D/3D, bc, color1, mis, pagerank1/3, sssp, our ray tracers, bfs3/4, heartwall, hotspot, lud, nw, huff, dwt2d and ResNet. PPT, however, performs better for cor, gemver, gesummv, adi, fdtd2D, jacobi1D/2D, color2, pagerank2, bfs1/2, pathfinder, sc, backprop and LSTM.

Summarizing, we found our cache hit rate approximation technique to be improved significantly, especially for the L1 cache. For our ray tracing implementations—which were the initial reason for implementing our own simulator—our simulator improved significantly with a mean absolute percentage error of around 4 % for the L1 cache compared to errors around 10 % for PPT, which applies also analogous for the L2 cache. Our model outperforms PPT, because we use a model that is closer to the actual hardware and uses documented and previously estimated parameters of caches to model them. Our model does not make use of the SDCM approximation and rather uses implementations of the cache mechanisms itself in order to estimate if a memory reference was a hit. Additionally, we evaluate each step of our profiler separately—inclusively memory reference extraction, coalescing and address transformations—in order to ensure that our results have small derivations.

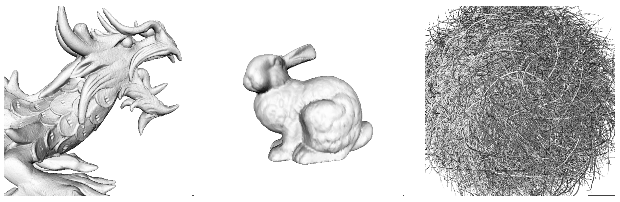

**(a)** *Dragon* **(b)** *Bunny* **(c)** *Hairball*

**Figure 3:** *Images from used meshes at the rendered perspective.*

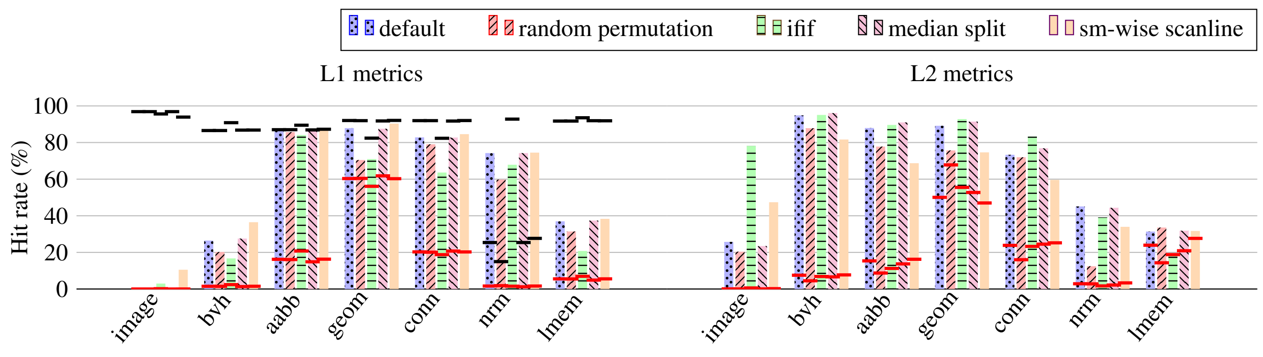
**Run-time Performance** Despite the comparison may be unfair, we measured run-time performance of profiling our ray tracer on the Dragon mesh and compare it against Nsight Compute and PPT for a rough assessment of computational time requirements. The running time varies between programs and a comparison mainly depends on the amount of memory accesses extracted. Nsight Compute suffers from multiple passes it executes for the same kernel but can make use of hardware performance counters. On a $512 \times 512$ view of the Dragon, our pipeline (1.02 min) is more than 4 times faster than PPT (4.75 min) but approximately 2 times slower than Nsight Compute (0.47 min). Pagerank looks different, where our implementation, as well as Nsight, takes 2 s and PPT 10 s. On the bicg benchmark, Nsight only requires 2 s, ours 14 s and PPT 55 s. We conclude, that we are approximately 4 times faster than PPT and usually slower than Nsight, which is hard to beat due to its use of internal hardware counters. However, Nsight Compute is not able to show per-allocation hit rates.

In the following, we use our previously presented accurate memory model to further narrow it down to individual allocations, by evaluating the outcome of eq. (3), instead of evaluating it per kernel call as in eq. (4).

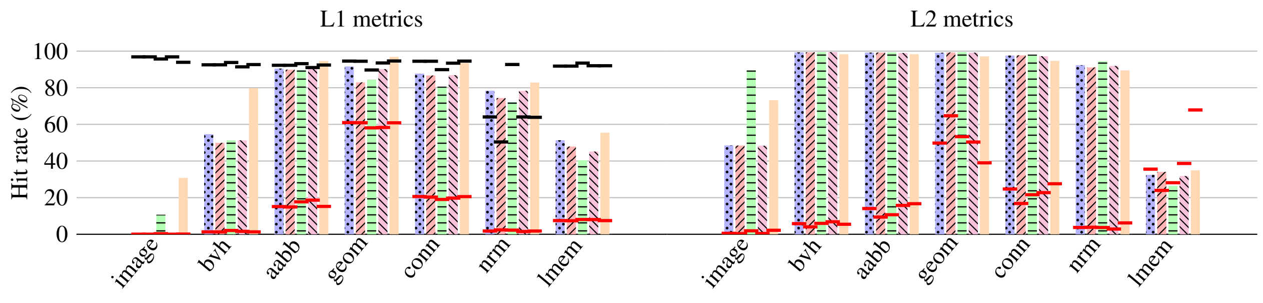
## 5.2. Fine-Grained Memory Profiles

We employ our fine-grained memory profiling onto our ray tracing implementation, as they were the initial reason for our fine-grained memory profiler and they contain a big set of actively used allocations within one kernel call due to the nature of ray tracing. We additionally shortly demonstrate our fine grained cache analysis on the neural network ResNet.

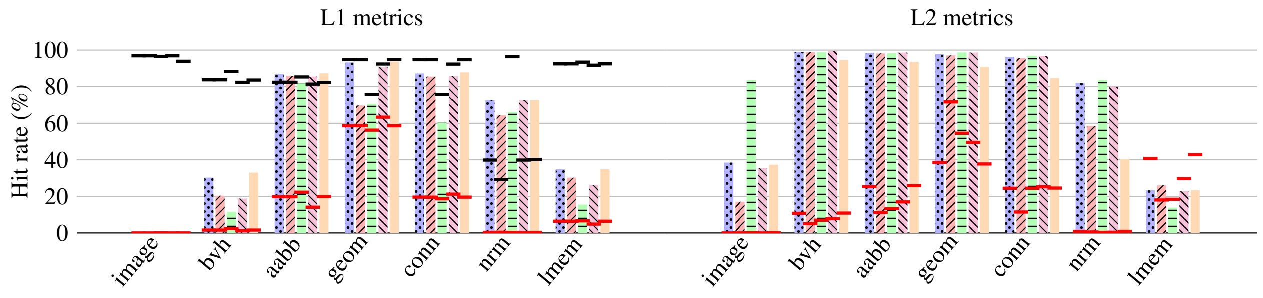
The ray tracer itself is split up into five similar implementations with different optimizations in order to allow comparison between different scenarios. We call these different implementations *configurations* in the following. The basis configuration (*default*) forms the *while-while* approach of AILA and LAINE [AL09] with persistent threads and a SAH-based generated BVH stored in a layout similar to the ray tracer of WALD, SLUSALLEK, BENTHIN, and WAGNER [WSBW01]. The second one (*random permutation*) applies a random memory permutation onto the mesh geometry that is expected to be less efficient than the status quo, as memory coherency obviously decreases. The third configuration (*ifif*) implements the analogous *if-if* work distribution, followed by replacing SAH with the median split strategy (*median split*) that is also known to be less efficient in most cases. The final configuration (*sm-wise scanline*) exploits the persistent thread approach



**(a)** *Dragon*



**(b)** *Bunny*



**(c)** *Hairball*

**Figure 4:** *Fine-grained profiling applied to our example tay tracer. Multiple metrics are superimposed in this diagram. Bars represent cache hit rates. Red lines stand for the relative number of memory transactions and black lines (in all cases above the red line) represent the coalescing efficiency.*

by distribute works in a non-standard way that neighboring pixels are processed by the same streaming multiprocessor, which we expect to perform better on on-chip caches like the L1 cache. We execute this set of ray tracing configurations onto a set of individual meshes. Precisely, onto the Dragon mesh from XYZRGB (fig. 4a), containing approximately 12 million triangles, onto the Bunny from the Stanford 3D scanning repository (fig. 4b), containing less than 70 000 triangles, and finally on the Hairball from NVIDIA Research (fig. 4c) with approximately three million triangles. The position and orientation of the camera can be seen in fig. 3.

Figure 4 shows our fine grained cache hit rate profiles on our meshes. Our generated diagrams visualize three metrics at the same time: cache hit rates for each allocation separately, the coalescing efficiency (i.e. the laod/store efficiency) and the relative memory traffic (i.e. the relative number of memory transactions). The latter in order to demonstrate that some allocations may have a marginal effect on the overall cache hit rate (see eq. (4)) and therefore also, potentially, on the run-time performance. The frame buffer and vertex normals are commonly such cases. Their memory traffic is far below 5 % in all our experiments. This is hardly surprising, as they are involved once per pixel. All other memory operations are inside the BVH traversal loop making memory references more frequent. The general trend is that the geometry of vertices receive most traffic and approximately three times more than the faces. This is also hardly surprising, as a single face (i.e. three vertex indices) is three times smaller than its three corresponding vertices in our ray tracing implementation. Despite their crucial importance for preventing

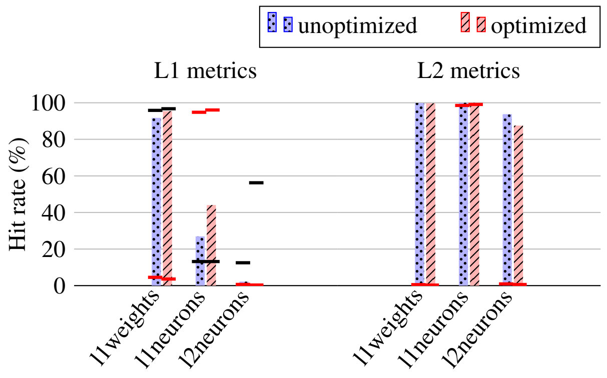


**Figure 5:** *Fine-grained profiling applied to the first kernel of ResNet from the Tango benchmark "executeFirstLayerCUDA". Multiple metrics are superimposed in this diagram. Bars represent cache hit rates. Red lines stand for the relative number of memory transactions and black lines represent the coalescing efficiency.*

faces to be accessed, AABBs seem to contribute less to the overall amount of memory references. For coalescing efficiency, we see high values for each allocation, except for the normals, which, however, occur infrequently. Accesses to local memory have an proportion of less than 10 % to the memory traffic. For the L2 cache, we see a higher variation in the relative number of memory transactions. This can be explained by their dependency on the L1 cache performance that we described in section 4.3.

**Random Vertex Permutation** More interestingly, all comparisons show that the random vertex permutation configuration decreases hit rates on the mesh geometry as well as vertex normals. This confirms our expectation that this ray tracer configuration affects caching on vertex-related allocations. Additionally, the caching performance on the local memory and the BVH tree decreases on all our testing datasets for this configuration.

**Work Distribution** Changing to the *if-if* work distribution has negative caching implications on all allocations, which corresponds to its slightly less efficiency [AL09]. The SM-wise scanline scheduling shows the expected trend. L1 hit rates on all allocations for all meshes as we increased SM-wise memory coherence. Nevertheless, the memory coherence appears to decrease globally, as caching performance decreases for the L2 cache.

**Median Split** We see an increased relative number of memory accesses to the mesh geometry and connectivity, especially for the Hairball. At the same time, caching in the L1 cache tends to perform slightly worse than our default configuration. We explain these inferior profiling values by the increased end-point overlap [AKL13] for median-split-based BVHs.

**ResNet** Figure 5 shows our profiler on the kernel that computes the first layer of ResNet as an example highlighting the potential of our tool outside of graphics algorithms, such as ray tracing. Our fine-grained profile reveals that caching on the input neurons is poor. This affects the performance of that program drastically, as the input neurons have the most memory traffic, which is also reflected in the diagram. A first analysis on usage of the corresponding allocation indicated that its indexing might be an factor of inefficiency for caching. We tried fixing this by swapping *x* and *y* coordinates—by just swapping two characters in the source code—in thread indexing, which effectively only decides if work is assigned row-wise of column-wise to a warp. Our memory profile then showed an increased cache hit rate for the neurons. On the RTX 2080 Ti, we measured a execution time of 2.99 ms on the same dataset as in our global evaluation without our optimization. With our simple optimization, we measure 1.35 ms.

## 6. Conclusion

Memory profiling is an important task in program optimization and requires in-depth knowledge of the hardware domain. In this paper we presented a fine-grained memory profiler that simulates the whole memory flow and finally accumulates profiling values in a way that the user retains information about the potential region in the profiled GPU program by showing them separately for each allocation.

In order to evaluate our memory profiler, we initially compared it to the official general purpose CUDA profiler *NVIDIA Nsight Compute* and a state-of-the-art memory simulator *PPT-GPU-Mem*. Results show that our model archives a mean error of 3.73 % and outperforms PPT by a factor of 2.4 for the L1 cache. L2 cache hit rates can be approximated with an error of 15.81 % with our tool, outperforming PPT by a factor of 1.3.

In a second step, we demonstrated our profilers capability of allocation-wise hit rate estimation by applying it on a set of different configurations of a ray tracing implementation. We found these results to correspond with initial assumptions on those configurations. Additionally, we showed how we successfully used our profiler to enhance the run-time performance of the first layer of ResNet from the Tango benchmark.

**Future Work and Limitations** In future, we would like to further improve our memory model to become even more accurate. Despite our memory profiler can help identifying potential sources of inefficiencies easier compared to bare whole-program cache hit rates, caching on individual allocations is still not independent on those and accesses to one may influence the cache rates of the other one negatively.

**Source Code** The source code for this paper is available at `https://github.com/maxvonbuelow/profiner`.

**Acknowledgements**

Part of the research in this paper was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 407 714 161. We thank the anonymous reviewers whose comments helped improve this manuscript.

## References

[ABC*20] ARAFA, YEHIA, BADAWY, ABDEL-HAMEED, CHENNUPATI, GOPINATH, et al. "Fast, accurate, and scalable memory modeling of GPGPUs using reuse profiles". *Proceedings of the 34th ACM International Conference on Supercomputing*. ICS '20. ACM, June 2020. DOI: 10.1145/3392717.3392761 1, 2.

[ACB*19] ARAFA, YEHIA, CHENNUPATI, GOPINATH, BARAI, ATANU, et al. "GPUs Cache Performance Estimation using Reuse Distance Analysis". *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*. IEEE, Oct. 2019, 1–8. DOI: 10.1109/ipccc47392.2019.8958760 1.

[AHH89] AGARWAL, A., HENNESSY, J., and HOROWITZ, M. "An analytical cache model". *ACM Transactions on Computer Systems* 7.2 (May 1989), 184–215. DOI: 10.1145/63404.63407 1.

[AKL13] AILA, TIMO, KARRAS, TERO, and LAINE, SAMULI. "On quality metrics of bounding volume hierarchies". *Proceedings of the 5th High-Performance Graphics Conference on - HPG '13*. HPG '13. ACM Press, 2013. DOI: 10.1145/2492045.2492056 2, 8.

[AL09] AILA, TIMO and LAINE, SAMULI. "Understanding the efficiency of ray traversal on GPUs". *Proceedings of the 1st ACM conference on High Performance Graphics - HPG '09*. HPG '09. ACM Press, 2009. DOI: 10.1145/1572769.1572792 2, 6, 8.

[BYF*09] BAKHODA, ALI, YUAN, GEORGE L., FUNG, WILSON W. L., et al. "Analyzing CUDA workloads using a detailed GPU simulator". *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, Apr. 2009, 163–174. DOI: 10.1109/ispass.2009.4919648 1.

[CBM*09] CHE, SHUAI, BOYER, MICHAEL, MENG, JIAYUAN, et al. "Rodinia: A benchmark suite for heterogeneous computing". *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Oct. 2009, 44–54. DOI: 10.1109/iiswc.2009.5306797 2, 4.

[CBRS13] CHE, SHUAI, BECKMANN, BRADFORD M., REINHARDT, STEVEN K., and SKADRON, KEVIN. "Pannotia: Understanding irregular GPGPU graph applications". *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Sept. 2013, 185–195. DOI: 10.1109/iiswc.2013.6704684 2, 4.

[DZ03] DING, CHEN and ZHONG, YUTAO. "Predicting whole-program locality through reuse distance analysis". *ACM SIGPLAN Notices* 38.5 (May 2003), 245–257. DOI: 10.1145/780822.781159 1.

[Fly66] FLYNN, M.J. "Very high-speed computing systems". *Proceedings of the IEEE* 54.12 (1966), 1901–1909. DOI: 10.1109/proc.1966.5273 2.

[GXS*12] GRAUER-GRAY, SCOTT, XU, LIFAN, SEARLES, ROBERT, et al. "Auto-tuning a high-level language targeted to GPU codes". *2012 Innovative Parallel Computing (InPar)*. IEEE, May 2012, 1–10. DOI: 10.1109/inpar.2012.6339595 2, 4.

[JMSS19] JIA, ZHE, MAGGIONI, MARCO, SMITH, JEFFREY, and SCARPAZZA, DANIELE PAOLO. "Dissecting the NVidia Turing T4 GPU via Microbenchmarking". (2019). DOI: 10.48550/ARXIV.1903.07486. Pre-published 3.

[KJ10] KHAN, SAMIRA and JIMENEZ, DANIEL A. "Insertion policy selection using Decision Tree Analysis". *2010 IEEE International Conference on Computer Design*. IEEE, Oct. 2010, 106–111. DOI: 10.1109/iccd.2010.5647608 3.

[KKS*19] KARKI, AAJNA, KESHAVA, CHETHAN PALANGOTU, SHIVAKUMAR, SPOORTHI MYSORE, et al. "Detailed Characterization of Deep Neural Networks on GPUs and FPGAs". *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs - GPGPU '19*. GPGPU '19. ACM Press, 2019. DOI: 10.1145/3300053.3319418 2, 4.

[KSAR20] KHAIRY, MAHMOUD, SHEN, ZHESHENG, AAMODT, TOR M., and ROGERS, TIMOTHY G. "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling". *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, May 2020, 473–486. DOI: 10.1109/isca45697.2020.00047 1.

[MB90] MACDONALD, J. DAVID and BOOTH, KELLOGG S. "Heuristics for ray tracing using space subdivision". *The Visual Computer* 6.3 (May 1990), 153–166. DOI: 10.1007/bf01911006 2.

[MC17] MEI, XINXIN and CHU, XIAOWEN. "Dissecting GPU Memory Hierarchy Through Microbenchmarking". *IEEE Transactions on Parallel and Distributed Systems* 28.1 (Jan. 2017), 72–86. DOI: 10.1109/tpds.2016.2549523 2.

[NvdBCB14] NUGTEREN, CEDRIC, van den BRAAK, GERT-JAN, CORPORAAL, HENK, and BAL, HENRI. "A detailed GPU cache model based on reuse distance theory". *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Feb. 2014, 37–48. DOI: 10.1109/hpca.2014.6835955 1.

[NVI22a] NVIDIA CORPORATION. *CUDA Programming Guide*. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide 2.

[NVI22b] NVIDIA CORPORATION. *PTX: Parallel thread execution ISA*. 2022. URL: https://docs.nvidia.com/cuda/parallel-thread-execution 3.

[TYL11] TANG, TAO, YANG, XUEJUN, and LIN, YISONG. "Cache Miss Analysis for GPU Programs Based on Stack Distance Profile". *2011 31st International Conference on Distributed Computing Systems*. IEEE, June 2011, 623–634. DOI: 10.1109/icdcs.2011.16 1.

[VSM*18] VASIOU, ELENA, SHKURKO, KONSTANTIN, MALLETT, IAN, et al. "A detailed study of ray tracing performance: render time and energy cost". *The Visual Computer* 34.6-8 (Apr. 2018), 875–885. DOI: 10.1007/s00371-018-1532-8 2.

[VSNK19] VILLA, ORESTE, STEPHENSON, MARK, NELLANS, DAVID, and KECKLER, STEPHEN W. "NVBit. A Dynamic Binary Instrumentation Framework for NVIDIA GPUs". *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. ACM, Oct. 2019. DOI: 10.1145/3352460.3358307 1, 2.

[WMZ22] WALD, INGO, MORRICAL, NATE, and ZELLMANN, STEFAN. "A Memory Efficient Encoding for Ray Tracing Large Unstructured Data". *IEEE Transactions on Visualization and Computer Graphics* 28.1 (Jan. 2022), 583–592. DOI: 10.1109/tvcg.2021.3114869 2.

[WSBW01] WALD, INGO, SLUSALLEK, PHILIPP, BENTHIN, CARSTEN, and WAGNER, MARKUS. "Interactive Rendering with Coherent Ray Tracing". *Computer Graphics Forum* 20.3 (Sept. 2001), 153–165. DOI: 10.1111/1467-8659.00508 6.

[WSWG13] WODNIOK, DOMINIK, SCHULZ, ANDRE, WIDMER, SVEN, and GOESELE, MICHAEL. "Analysis of Cache Behavior and Performance of Different BVH Memory Layouts for Tracing Incoherent Rays". *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2013. DOI: 10.2312/EGPGV/EGPGV13/057-064 2.

[WX16] WANG, DONGWEI and XIAO, WEIJUN. "A reuse distance based performance analysis on GPU L1 data cache". *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*. IEEE, Dec. 2016, 1–8. DOI: 10.1109/pccc.2016.7820638 1.