Massively Parallel Program Analysis Using Simulation of Graphics Processing Units

Dmytro Rahozin [1]

[1] *Institute of Software Systems of the National Academy of Sciences of Ukraine, Acad. Hlushkov ave. 40, Kyiv, 03187, Ukraine*

Abstract

Modern Graphics Processing unit (graphics card) is a modern platform to run massively parallel programs. Still, it is a complex task to analysis, observe and measure performance of GPU-based software. Due to complex memory hierarchy, specific organization of memory units and the use of thousands of execution threads the performance improvement is a task about the efficient use of graphics card memory hierarchy. We describe the use of GPGPUSim simulator, previously used mostly for graphics card architecture validation, for performance analysis for CUDA-based program, which covers many performance analysis questions for parallel software. We provide examples which show how to use the simulation for performance analysis of massively parallel programs.

Keywords

Graphics processing unit, software performance, massive parallelism, simulation, software performance model.

1. Introduction

Over last 15 years graphics processing units (GPUs) or video-cards have revolutionized high performance computing, introducing massive parallelism at a moderate price and allowing everybody to have a supercomputer at home. Still the effective use of a GPU is limited as the efficient GPU software creation usually requires the full redesign of the original algorithm, as GPU provides quite a different model of massive parallelism utilization.

First, we discuss the existing GPU performance models, which are based on software simulators of GPU hardware. This software is able to simulate computational units, GPU memory hierarchy model, commutation network and are able to execute usual programs compiled for GPU, for example PTX codes of CUDA toolkit for Nvidia GPU. The most simulators employ the reengineered GPU architecture [1], which was recovered using various benchmarks [2], so allows the end-user to be aware about better techniques of software optimization for video-cards.

Second, first GPU where quite limited computational devices, targeted for pixels rendering, triangle meshes calculation and texture rendering. In 2007, Nvidia 8800 GPU revolutionized the market, as Nvidia GPUs provide a massive parallel programming model, giving the user a massive parallel processor. But the internal problem for programming model was not a complexity of programming model, but just a being unprepared for introduction of such parallel programming devices on market. Year by year the GPUs became more and more “general-programming ready” at the cost of increasing the complexity of memory buses, but providing 1000s of computational cores. Multicore CPUs performance was not scaled so much, due to lack of flexibility at memory side. The use of GPU for computations is still too expensive in programming (time/money) for software developers, but its architecture looks to be familiar for “supercomputer” user – except challenges with double precision floating point, the GPU platform looks familiar for people who used to operate with thousands of threads.

[1] 13th International Scientific and Practical Conference from Programming UkrPROGP’2022, October 11-12, 2022, Kyiv, Ukraine
EMAIL: Dmytro.Rahozin@gmail.com (A. 1)
ORCID: 0000-0002-8445-9921 (A. 1)

© 2022 Copyright for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-3501/s5.pdf

Third, there is a problem of the inefficient utilization of GPU massive parallelism, so that for complex software it is hard to provide the good higher-level model of massive parallelism, which can be effectively projected onto the GPU hardware [3]. It is hard to account the relations and dependencies of the basic parameters of software massive parallel model – for example, number of threads, memory block size processed by one thread, scheduling techniques and so predict the final real performance. GPU computing model was really revolutionized the computing and we state that the GPU performance is fully defined by memory hierarchy performance. As the GPU memory hierarchy employs 5-6 memory levels now (compares to the older L1-L2-L3 cache memory scheme) the analysis of bottlenecks caused by memory hierarchy/interconnect is extremely complex. Definitely the software developer needs good tools which help him to analyze what is wrong with software performance and need to define simpler model for software performance.

Fourth, we propose the use of simulator to discover the potential problems in massively parallel software, analyze the software bottlenecks and recover memory utilization models. We re-formulate the performance analysis problem as a problem of the different memory hierarchy stages resource utilization, it is very close to the commonly used mass service models. The role of simulator is changed – originally the simulator was used for recovering the peculiarities of a GPU architecture, but now we are extracting model data from memory hierarchy simulation. The simplified model is applied back to the original task and allows to estimate memory/time resources necessary for software run.

Fifth, we are providing the motivating example of CUDA application analysis on the top of simulation and illustrate how the simulation data can be used to recover less complex performance model and re-applied for application optimization. We are providing suggestions for simulator enhancement and enabling better performance analysis, leading to build-up of clear performance estimation of the developer massively parallel software.

2. GPU performance models

Currently we are observing several shifts in microprocessor production, and these shifts hugely impact existing approaches for application development. First, the term “parallel application development” became more “academic”, as the process of researching different forms of parallelism in application became even more complex, even more time-consuming and requires a lot of knowledge in narrow areas. Second, modern “start-up” culture pushes engineers to decrease “time-to-market” and development costs for their products or product prototypes, so the employment of newer and newer architectural capabilities become slower and slower and comes down to moving just to newer GPUs or running several copies of a process on more powerful GPU. The real things a “median” programmer can do around “parallelism” utilization is the employment of threading concept without good understanding of resulting application performance. This gives a chance to chip makers to increase the number of processor cores on chip, as they easily can be employed in applications or operating system by just run usual software services. So basically, applications are not optimized for multi core CPUs. Further performance analysis and optimizations are open to industrial rivals, who can enter the market with a substitute application, which is faster and smarter. So, the scaling of the number of processor cores looks to be a good choice for typical smartphone platform which runs a bunch of application which are loosely coupled in terms of using some data amounts simultaneously.

Although if we look into software, which we can define as “data processing infrastructure” – communication, video processing, artificial intelligence (AI) [4], we see the opposite picture – each year the volume of processed data increases exponentially. For example, a century ago, a short telegram message was enough for communications, 20 years ago we entered the era of voice conferences with sophisticated sound processing, and now we demand video calls and conferences as we need the visual presence of our peer. Nobody can predict the next shift (mostly a move to virtual reality area) where computation demand will increase another 10x.

This increase of computing demand targets not the computation capability and not the memory volume – as the computational cores and memory cells are scaled perfectly, but the interconnect communication, which joins storage and computations (fig 1.). It’s impossible to scale interconnect infrastructure due to physical limitations, but everybody notices that fig. 1 illustrated the usual computational system structure – it may be applied to microcontrollers, common multicore system

and GPU. Actually, this means that all the computer system has the same problem of non-scaling interconnect network, but today the interconnect scaling limitation became at least “severe”. Note, that GPU hardware has the best current solutions for optimizing the interconnect, as it requires to process huge data (triangles, texture and points) in complex way (rendering, ray-tracing, general computing) in real-time (60 frames per second).

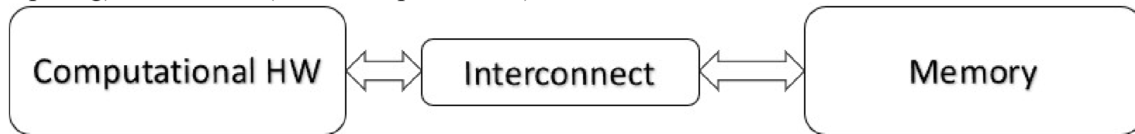

Figure 1: Computational system structure.

Pure computation capability is practically infinite now, the number of memory cells is really huge, but the interconnect between them – as the data is routed to computational units - is very limited. Even more limiting factor is the complexity of interconnect programming by a developer. There is a good (and ancient) example for this – older DSPs has several independent memory buses, as only this solution doubles or triples the data throughput. The technology required the special programming, as program data are spread among two or more memory spaces manually. Now there are more complicated chips with multilevel cache memory but the major problem is still the same – memory is quite slow if compared to computations, so channels capacity between logic and memory should be increased, and here it looks that it is necessary to change the paradigm. If we check modern GPUs (as of 2019), they have 12 (twelve) or more high-speed (DDR4/5 DRAM) controllers, which should (at least in theory) balance memory load/throughput.

Long ago a simple but practical classification for software performance limitations was introduced: memory-bounded software and computations performance bounded software. The latter case was resolved quickly usually in two ways: 1) computations are scaled via SIMD execution and just providing extra number crunching power (but need to scale memory buses); 2) common pipelines are defined in hardware and used as a hardware accelerator (e.g. Nvidia video stream encoders and decoders). The memory-bounded case is much harder - memory bus became wide, up to 2048 bits – to feed SIMD units width or hardware accelerators, but never bus speed can outperform arithmetic speed – even worse multi-core system just multiple memory load. Basically, if we are taking out of view the small number of really computational bounded algorithms, the most important characteristic of a program/algorithm is the degree of “memory boundness”. Overcoming the memory bandwidth limitations requires the improvement of both hardware and software methods, as the interconnect infrastructure should be load optimally.

The interconnect does not mean just a bunch of wires [3]. Computational units produce requests for reading/writing memory cells, memory is an old-fashioned DRAM with need to open pages, refresh contents and serve operations. Basically, the pseudo-random memory accesses greatly decrease the DRAM memory throughput, so interconnect should effectively decrease the number of memory accesses exploiting the principle that the data traffic for the most frequently used memory cells should be routed somewhere inside the interconnects. Speaking simply, interconnect should integrate specific multilevel cache memory to serve multi-threaded execution, which multiplies memory traffic.

As a good example of the threading paradigm change, we address the execution structure of GPU hardware threads – where the massive thread execution is driven by readiness of data, fetched from interconnect internals. The advanced approach requires simultaneous execution of hundreds of threads. Some of them processing data, some of them waiting for data to be ready. This raises at least two common problems: 1) an algorithm should have enough degree of parallelism to allow execution of hundreds of threads – for example, triangle and pixel rendering; 2) an algorithm should be initially developed to utilize the benefits of memory hierarchy (interconnect). The first problem is inevitable, as somebody need to convert an initial task into a concept of a set of interoperating threads. The second problem is more technical, but we need to define the interconnect and memory hierarchy structure in computing systems. For practical reason we state that the interconnect and memory hierarchy need to be architected to support hundreds of threads, conceptually this is very different from simple interconnects of multicore processors.

The rise of cheap mobile system-on-chips (SoCs) leads us to new challenges in massively parallel computing (really supercomputing), proving excessive parallelism level but making code optimization time consuming and costly. A mobile GPU with thousands of execution threads is a flagship of modern computing and the possible level up looks to be a kind of quantum computer, so software development industry has to invent ways to utilize this computing power efficiently.

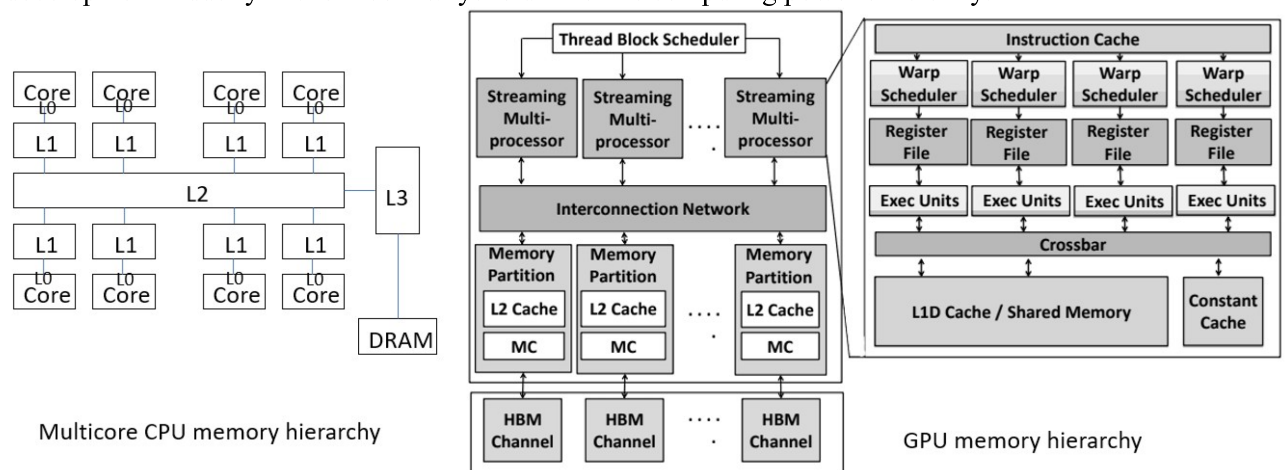

Figure 2: Memory hierarchies for multicore CPU and GPU [1].

Going back to traditional multi-core CPUs we consider the scheme (fig. 2, left) of multilevel cache memory (or interconnect), which usually include fast but very small L0 memory (read/write queues to L1 cache memory), faster L1 cache memory, and L2 and L3 cache memories, L2 and L3 look to be quite slow but really big. For streaming applications, we should also account DRAM performance, so current multicore CPU memory hierarchy includes up to 5 hierarchy levels, and 95% of memory bandwidth optimizations are around the better use of L1/L2 cache. More optimizations for memory hierarchy lead to exponential growth of optimization costs. Optimizations for L1/L2 cache improve data spatial locality but the other memory hierarchy is not taken into account due to too much cost and still the number of hardware threads is small. The fig. 2, right shows us much more complicated interconnect for only one core at modern GPU. The fig. 2 does not show several hundred threads on-the-fly, generating requests for memory. These threads involve the use of the following hardware (fig. 2, right): 1) cross-bar interconnects/coalescers, allowing gathering/scattering data or connecting cores to different L2 parts; 2) "non-blocking" L1 streaming cache, which eliminate stall during memory accesses; 3) Specific schedulers for multichannel DRAM access. And the huge number of executing threads require specific programming methods to utilize the hardware efficiently. Let us discuss the motivating case.

3. Programming model for massively parallel programs

The usual first question people have about GPU programming is the question of proper abstraction over the exposed massive parallelism, we call it a "programming model" for our purposes.

Let's start not at supercomputing but at "microcomputer" point, such as old good 8088 CPU. The "thread" term as a basis for programming model was appeared long before the first dual-CPU machine was delivered for end user. The representation of a program as a bunch of interoperating threads allows at least to incorporate several thread-enabled programs into a complete software system. The ability to run a thread-enabled software on a multiprocessor and have the shrinking of execution time is a result of the proper expression of the available parallelism in terms of the programming models. Sure, the developer should adopt the concept of synchronization points, be aware of common principles of parallel programming and be able to extract natural parallelism sometimes even using a pencil and paper.

The most primitive parallel programming models are *pthreads* and OpenMP model. The first model is widely used when you just need a threading model, possibly with manual control over thread running and priority; the second is used for simplest and obvious parallel applications. There is no

notion of underlying memory model, cache memory hierarchy and so on, but this works good enough for older double or four processor computers. Initially general-purpose computers have the simple memory architecture, with tendency to increase and improve cache memory and so hide memory latency. Simpler multicore model fits still well modern user programs within *pthread* model, when, for example, a text editor allocates one core and an accounting program allocates another core. This model scales well up to 16 cores, scaling for more cores raises the same problems for parallel applications, as we have for x86 computers such as Xeon Phi [11]. The increase of the number of cores leads to the complexity increase of memory architecture and to the same parallel programming problems as we have for existing GPU regardless of computational power. Looks like there is a border between parallelism and "massive" parallelism, highly related on memory hierarchy complexity

Meanwhile GPU architecture evolved just in the opposite direction: originally it has quite complex programming and memory model – old school developers still remember programming model based on vertex and pixel shaders. Nvidia 8800 card was the first which provides a kind of more familiar "general-purpose" programming model. The "newly" introduced CUDA technology allowed to run "general-purpose" programs with some limitations. But the cornerstone of the parallel programming was the effective use of available parallel resources, which include SIMD-fication, warp-aware programming, specific memory allocation for computational units, specific synchronization. The exposed CUDA programming model was the tradeoff between general purpose programming and requirements for efficient pixel and vertex shader runs. Of course, the programmer may not use the GPU hardware, such as SIMD, threads and may not be aware of proper parallelization techniques, in this case the 99% of GPU computational power will be wasted. Anyway, existing GPU architecture looks to be well-balanced, as Nvidia neural networking accelerator card, which have no video output, has the same hardware architecture as common GPUs, so even removing the requirement to run graphics-related shaders does not change the state of art in existing GPU programming model.

But the convergence of GPU and multiprocessor programming models is not possible in foreseeable future, mostly due to the difference in concepts, as GPU involves programmer heavily into architecture details. Moreover, if the massive threading is evident for CUDA, the internals of memory hierarchy are unclear. Still the performance analysis for GPU software and memory footprint optimization are the area for the small number of computer scientists.

GPGPUSim [2] itself uncovers the internals of GPU card for several areas of interest. First, as it does partial work of a GPU driver, so you can look into mechanisms which run a CUDA application on GPU. Second, an attentive programmer can get well enough information about proper GPU use and GPU capabilities from the simulator code. Despite the common sense that GPU architecture itself is very complex and somehow mysterious, all the hardware GPGPU computing concepts are derived from public articles in computer science journals, as any architectural advances in computing should be simulated on a hardware simulator similar to GPGPUSim before going into hardware. Speculating, it is possible that GPUGPUSim is used by real chipmakers to check architecture innovations.

Third, GPUGPUSim simulator has the smaller simulator inside, internally called InterSim2. It simulates a general routing network, with extreme number of options, which enable to simulate practically all types of packets routings and scheduling methods, described in scientific papers. The interconnect is used to connect partitioned caches, for example L2 GPU cache, forming a holistic L2 cache level, which is perfectly integrable into memory hierarchy. In our opinion the roots of this interconnect fabric are situated in older supercomputers (even from 70s), so are good analyzed and tested on many workloads, so that these ideas were the good points to start analysis for GPU interconnect development. This structure is much more complex than ring bus used in Xeon Phi from Intel, but Xeon Phi has much smaller number of CPUs sitting on bus. We are not going here to compare performance of Xeon Phi [11] and GPU due to different programming models for Xeon Phi and GPU and the limited number of CPUs on Xeon Phi bus. So we can not project the real bus performance of the ring bus for massive parallel tasks. Anyway, it should be noted that the interconnect is the only one of many architectural solutions enabling massive parallelism on GPU.

Finally, let's return to the programming model philosophy for GPU. It is obvious, that it quite hard to make conclusions over simulating CUDA programs in global sense, due to quite low-level coding in CUDA. It looks more useful to adopt more high-level software, for example cuDNN library, as the developer already was moved level up from linear structures up to 2D structures and it is much easier to compare higher-level memory operations (such as evaluating a level in a neural network) and

updates in simulation results. Here cache effects after changing 2D data structures looks clearer. Looks like that using high-level languages like Haskell/APL, where 2D array slicing is used for coding directly on GPU can shorten the way between defining 2D array slices operations and analysing memory bandwidth utilization effects in simulator. But this is the next points in our research.

4. Software performance modeling

Despite of the wide use of matrix computation examples in performance-related studies, still BLAS/LAPACK based software is used for solving many problems (or BLAS/LAPACK style of expressing matrix operations), e.g. for signal/sound processing area. Convolutional neural networks software uses special optimized routines for matrix processing, but general matrix-processing based software still use LAPACK due to old and well-known standard. Nvidia proposes cuBLAS library for GPU-optimized BLAS routines.

C_{11}	C_{12}	C_{13}	C_{14}		A_{11}	A_{12}	A_{13}	A_{14}		B_{11}	B_{12}	B_{13}	B_{14}
C_{21}	C_{22}	C_{23}	C_{24}	=	A_{21}	A_{22}	A_{23}	A_{24}	*	B_{21}	B_{22}	B_{23}	B_{24}
C_{31}	C_{32}	C_{33}	C_{34}		A_{31}	A_{32}	A_{33}	A_{34}		B_{31}	B_{32}	B_{33}	B_{34}
C_{41}	C_{42}	C_{43}	C_{44}		A_{41}	A_{42}	A_{43}	A_{44}		B_{41}	B_{42}	B_{43}	B_{44}

Figure 3: Illustrative block matrix computation scheme

The common matrix by matrix multiplication scheme is shown on fig. 3. In order to computer C matrix block, a block row from A should be multiplied by block column B. As above we have mentioned that the common software workloads are memory bandwidth bounded and this example is classical – for each multiplication operation which is performed in 1 clock cycle, we need 2 memory accesses. The compared energy consumption for multiplication (a small functional unit) and for memory access (memory, buses, interconnect, crossbar switches) is least 50x different. So, for speed and energy consumption all A_{2i} and B_{i2} should be synchronously loaded to L1D cache/shared memory (fig. 2, right) and possibly reused for computing C_{2i} blocks. Standard approaches for cuBLAS matrix multiplications optimization are described in [5], advanced autotuning result is presented in [6]. The autotuning approach [7] has many pros but a kind of contra – the tuned library is fast "in common sense" as the target parameter is time, and sometimes additional resource restrictions apply (e.g. less time – more memory or software depends on "hot cache") and a library may use very specific dataset, and this is common for the most of computational tasks – Fourier transforms have only one or two used dimensions, matrices have only several fixed dimensions. The important point is that several copies of computational process may be run on the same GPU (e.g. neural networks computations) to better utilize non-used computational resources.

Anyway, matrix multiplication is scheduled as an execution graph of smaller partial matrix-multiplication procedures of different sizes. There are various tradeoffs here, as complied kernels have pre-defined memory allocation layout and their parametrization (loop bound change) and sometimes need recompilation and re-scheduling and this is common for GPUs. The compiled multiplication kernels reside on SMT (Streaming Multi Processor, fig. 2, right), which has limited register file (local memory, usually 64K memory cells) which handles internal kernel variables and shared memory cells needed for information exchange between kernels. The compiler also should divide the memory between threads local storage and shared communication memory, that directly influences the possibility of kernel allocation on SMT – the more memory kernel uses for communication, the smaller number of threads are executed on the kernel – another tradeoff, highly depend on currently executed task set.

Initially the "big" multiplication algorithm includes an optimized scheme for data transfers between kernels to save memory bandwidth. Of course, kernels should be parametrized, for block sub-sizes and data type. Reaching some successful tradeoff includes finding out the optimal parameters for size of the general block matrix multiply kernel, by 1) adjusting the size of memory handling matrix values; 2) adjusting memory size used as shared to exchange data between kernels.

The optimal piece of matrix processed by kernels is a tricky question, in general case very common suggestions (e.g. register file size) are used for the basic matrix blocks size, as we need 1) to compute 2) to exchange data 3) to run as many threads as possible. For the kernel tuning there is a pre-defined knowledge base initially, which stores basic register file parameters, so the tuning algorithm have defined opportunities to use different matrix block sizes and possible limits for matrix dimensions. The scheduler program defines the set of semi-optimal parameters for the "common" matrix multiplication procedure for this particular GPU. The problem is solved now using a benchmarking system [], which runs multiplication with defined matrix block sizes, making the GPU driver to evaluate execution time. After that the final library is hard-compiled with the optimal parameters and is used further for number crunching. If the code is moved to another hardware, we need to re-evaluate and recompile the library. Basically, this is the way commonly used in industry for example, for BLAS/LAPACK [5]. Still, this type of software tuning looks to be a "black box" methos– input task parameters produce results, but there are no evidences how to line up parameters to have the semi-optimal execution for selected datasets. None of benchmarking system is able to make a model, which finds out relations between kernel parameters and execution time – as benchmark uses only execution time as optimal criteria.

The upcoming problem is that the block-based matrix multiplication is the comparatively simple task but investigated more than 50 years for available parallelism. The public codes for BLAS/LAPACK show that even basic task looks to be complex, it is not just compiled library code but a software system, which generate semi-optimal source code after benchmarking the system. If somebody is required to produce a brand-new code it is even hard to imagine how much efforts should be invested into creating a semi-optimal parallel code which can exploit parallelism on a massively parallel processor. Even if the task can be parallelized well, we need to implement parametrized kernel and define the factors which influence the software performance and used while resolving later tradeoffs.

5. Performance modeling for memory hierarchies

In this chapter the model of the internal architecture and memory hierarchy (interconnect) of off-the-shelf video cards is considered. Only commonly used Nvidia video cards are considered – due to availability of the model, as since the appearance of the pioneering Nvidia 8800 programmable video card, this platform is actively used for different kinds of computations, including very specific tasks, which use imprecise computations [8] to employ thousands of hardware execution threads for faster computations. Another point is that the video card is the general number crunching machine, which has very good tradeoff between the peak computation power and memory bandwidth. Finally, it is really important that there is a code base of less or more optimized algorithms, which can be analysed for efficiency of memory bandwidth use.

For our investigation we use GPGPUSim simulator version 4.0.0 [9], which reflects the contemporary complex architecture of the interconnect between GPU SMT (fig. 2) processors and memory system. Originally the simulator was developed for recovering the architecture of Nvidia GPUs, including SMT, energy model, full interconnect for SMT registers down to DDR memory [1], as practically all hardware solutions used for computations and memory traffic routing were discussed in research papers, e.g. [3], but the particular combinations of architecture solutions is still o trade secret of Nvidia. The GPGPUSIM simulation precision is proved by simultaneous run of benchmarks (e.g. Rodinia benchmark [2]) both on simulator and on a real GPU and comparing numbers, the results of comparison may be found in [2]. For our purposes we state that the difference of GPGPUSim simulation and real execution on GPU is insufficient for our research.

GPGPUSim uses "fake driver" concept for running the applications. The simulator uses the fact that all CUDA based codes are compiled into a form of so called PTX codes, which are very similar to the real processor instructions of Nvidia GPU. Internally the GPU driver, which runs a CUDA program on a GPU, translates PTX code into real GPU instructions, place on SMTs and execute it due to N-dimensional execution model used by CUDA. This execution model effectively hides the GPU internals from the user but gives the promise that well parallelized software will we well scaled for the next generation GPUs supporting even more threads.

GPGPUSim works as a driver and intercepts PTX code execution, so does the same work as CUDA translator and scheduler do, but only on CPU and on single thread. So GPGPUSim runs even on a computer without real GPU. The simulation process is highly time consuming – so the standard matrix multiply application with default parameters from CUDA samples package is run for two days under simulator. This is not a problem as an older computer fleet which is able to run Ubuntu 20/22 and GCC 7 toolchain can be used to simulate CUDA application runs. GPGPUSim provides different types of simulation – 1) just functional, 2) simulation with all instruction timings but zero memory timings, 3) simulation of the whole SMTs-memory interconnect including DDR5 memory simulation models. There is a special mode for energy consumption simulation. It should be noted that simulator codes are placed in GitHub, so can be freely downloaded and changed. There are several simulator extensions, e.g. Accel-Sim [10], which are used for precise simulation validation and analysis.

GPGPUSim is highly parametrized – more than a hundred of options control the simulated hardware blocks, from SMT to DDR memory. As the GPGPUSim authors struggle to have simulation accuracy in less than 5% if compared to real HW, all the hardware components are precisely simulated. For a simple example, practically all the timing values from DDR5 chips manuals describing DDR5 memory bus are accurately introduced into the memory model. All the specific concepts, which enable the huge parallelism level, for example, streaming L1 caches [3], are simulated, so that the application developer can observe the real effects of using the newer hardware concepts for the application. The simulated interconnect complexity is really high to account even a basic set of optimizations for the parallel code. The developer who wants to optimize the code should sit with “paper and pencil” and plan the memory layouts and scheduling of the parallel applications.

For our purpose we need to analyze a set of performance counter which are generated by GPGPUSim during the program execution. For the first glance these counters can be divided into two sets: 1) counters at the interconnect level borders; 2) counters for internal events in interconnect levels. Let’s consider fig. 4.

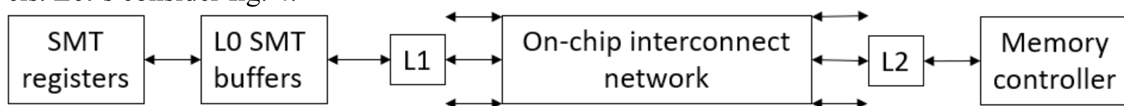

Figure 4: Memory hierarchy/interconnect levels.

Fig 4. includes the large blocks of SMTs to DDR memory interconnect/hierarchy [1]. Here arrows are “located” at the real hierarchy borders, and basically for each arrow we are able to denote the traffic from more upper hierarchy level to lower level (e.g. from L2 cache to memory controller). The efficiency degree for the hierarchy level for the particular task (except interconnect as it just routes the data between L1 cache banks and L2 cache banks) is the bigger difference of the data amount routed to the level from higher level (left at fig. 4) and data routed to the lower levels of memory hierarchy (right at fig. 4). Researches also should be interest in the data set sizes, which are more friendly to the hardware, as despite of increasing DDR memory size on board, the internal cache/shared memory sizes for SMTs is practically constant over GPU generations. The internal events counters are used for the fine tuning of the algorithms, as are able to show how different memory level improvements are employed by the parallelized version of algorithm, as different parallelization methods can utilize internal helpers in different ways. Here we are discussing “level borders” counters, the internal level counters are the subject for our next research.

6. Motivating examples

Now we propose a methodic, which starts from very beginning analysis of memory traffic. There are different levels of hardware observations and currently we starting from high level beginning metrics. So let us discuss what is valuable here for us and what are the challenges for research from the simulator side.

Each computational demanding application is started from computational model, which usually have incredibly low speed. For example, LIDAR model application has 1 frame per second (fps) computational speed, but the final requirement is 100 fps. So, the initial task for simulation is the evaluation of peak memory performance of the model application. Note, that the “computational

demand" here is transformed into "memory performance", as memory throughput is more limiting than computational throughput. L2 cache memory misses per computed frame describe the basic need for memory traffic well, and (without optimizations) helps to project upper memory bandwidth limit. Local optimizations for CUDA procedures may improve memory bandwidth up to 10x-20x, but the detailed information may be got from simulation run. Anyway, any performance model made for multicore CPU does not project real scaling for hundreds of threads and does not reflect the optimization effects possible on GPU. That's why massive parallelism simulation and evaluation are not possible on multi-core CPU as the difference in number of threads is up to 50-100x, so none of real parallelization issues may be observed at low number of threads. At high level of hardware observing a developer can notice the upper limits of memory traffic necessary to make a decision for feasibility of optimized program run at GPU, at lower level of observations it is possible to check even fine effects of different cache levels use.

For the entry level *matrixMul* CUDA sample application was used. To check the cache utilization effects the application was run (simulated) with different matrix size parameters, starting from 64x64 matrix and finishing with 256*896 size. The default multiplication size of 320x320 per 640x320 requires 2 days simulation on Intel Core i5-10400, at 1 thread, so the dataset configuration was adjusted to decrease simulation time and reflect real problem sizes.

The GPGPUSim by default generates a text file, which updates the internal simulation counters and model in timely manner. As the simulator works over the GPU state - the simulation results are functionally correct, and the program/shaders are executed correctly. The resulting log file can be parsed using any tools, starting from basic *grep*/*findstr* utilities up to complex Python scripts. In order to parse these log files, we used simple *findstr*-based scripts. Simulation was run on Ubuntu 22 OS, Intel 10th Gen based and equipped with 32GB RAM, but retargeted for GCC 7 toolchain, as other toolchains fall into compilation errors.

The simulation results are gathered into the following tables (fig. 5, fig. 6), we render only the most important simulation results. For each *matrixMul* run we collect matrix dimensions (for reference we note matrix sizes, just to know which data amount is processed during the run), L2 cache accesses, miss rate and parallel utilization of the L2 cache. L2 accesses mean the total data traffic we get from L1 cache. L2 misses mean the data traffic goes to DRAM. And the most interesting point is the L2 cache parallel utilization, the more this digit the better is the L2 sectored cache utilization, as L2 structure allows to handle accesses in different L2 sectors simultaneously. The see some imbalance in L2 parallel utilization (fig. 5) in 64x384x64, 128x768x128 and 256x768x256 multiplication – the hypothesis that memory layout is these cases is imbalanced and some L2 sectors are loaded much more than others.

N	K	M	L2 accesses	L2 misses	L2 miss rate	L2 parallel	Commands	Matrix size, KB
64	64	64	770560	512	0,0007	2,6751	8796931	16
64	128	64	1387008	512	0,0004	2,3871	13156851	32
64	192	64	2003456	512	0,0003	2,1245	17578119	48
64	256	64	2619904	512	0,0002	1,938	22032941	64
64	320	64	3236352	512	0,0002	2,3605	26071177	80
64	384	64	3852800	512	0,0001	1,6027	31004616	96
N	K	M	L2 accesses	L2 misses	L2 miss rate	L2 parallel	Commands	Matrix size, KB
128	128	128	5548032	2048	0,0004	4,8517	13789351	64
128	256	128	10479616	2048	0,0001	2,6501	23982502	128
128	384	128	15441200	2048	0,0001	2,0584	34300430	192
128	512	128	20342784	2048	0,0001	2,1442	41829619	256
128	640	128	25274368	2048	0,0001	2,8149	49129929	320
128	768	128	30205952	2048	0,0001	1,762	67740651	384
N	K	M	L2 accesses	L2 misses	L2 miss rate	L2 parallel	Commands	Matrix size, KB
256	256	256	40552548	8192	0,0002	2,7894	59223409	256
256	384	256	61644800	8192	0,0001	2,6323	86073012	384
256	512	256	81371136	8192	0,0001	2,7374	111265663	512
256	640	256	101097472	8192	0,0001	2,8753	136206534	640
256	768	256	120823808	8192	0,0001	2,4648	169508642	768
256	896	256	140550144	8192	0,0001	2,8143	188037216	896

Figure 5: Matrix multiplication simulation results

Bank	L2 accesses	L2 misses	Bank	L2 accesses	L2 misses	Bank	L2 accesses	L2 misses
0	5035128	344	8	5033924	340	16	5033924	340
1	5035128	344	9	5033924	340	17	5033924	340
2	5035128	344	10	5033924	340	18	5033924	340
3	5035128	344	11	5033924	340	19	5033924	340
4	5035128	344	12	5033924	340	20	5033924	340
5	5035128	344	13	5033924	340	21	5033924	340
6	5035128	344	14	5033924	340	22	5033924	340
7	5035128	344	15	5033924	340	23	5033924	340

Bank	L1 misses	Bank	L1 misses	Bank	L1 misses	Bank	L1 misses
0	4020352	8	4070528	16	3988992	24	4076800
1	4014080	9	4051712	17	4083072	25	4026624
2	4045440	10	4001536	18	4026624	26	4020352
3	4070528	11	4045440	19	4064256	27	4001536
4	4039168	12	4026624	20	3976448	28	3995264
5	4064256	13	4026624	21	4045440	29	3970176
6	4045440	14	3932544	22	3995264		
7	3951360	15	4089344	23	4057984		

Figure 6: L1/L2 cache load balancing

Fig. 6 shows the thread running balance in sense of the balance of memory accesses in sectored cache memory. The picture at fig. 6 is practically the same for any reviewed matrix size, so we see practically the ideal balance of L2 accesses and misses, this means that the matrices equally distributed over the existing processes. Please note that this balance works also for L1 cache, despite it has more sectors – 30 – if compared to 24 L2 sectors.

So, let us review what we can conclude for the "big" load picture for the matrix multiplication example. First, we see that the problem fits the cache in practically ideal way – the number of cache misses is less than 0.1% which is really small value. Yes, this should not surprise us as the matrix multiplication is the topic which is evaluated over 50 years, but such kind of analysis is usually provided for task which are unfamiliar for researchers. So, these results are extremely valuable if we have no prior knowledge about the problem. By the way, additional results are not included in the paper, but the same balancing analysis as in fig. 6 is computed for DRAM. As DRAM is multichannel (12 channels), in case of low L2 miss rates (more than 1-2%) a problem of proper parallelization of DRAM access arises, as 12 DRAM channels should effectively multiply DRAM throughput in case of correct parallelization. Anyway the fig. 6 data allow to understand load balancing at cache memory level and this metric directly highlights the quality of parallelization, which is ideal for our case.

7. Conclusions

Here we use GPGPUSim in the opposite way to the initial authors intentions – instead of re-engineering GPU architecture, we are evaluating CUDA-based software, as the simulation timing errors do not exceed 10%. This even is not important for our tasks, as we need information for memory accesses amount and patterns. For our classical task – matrix multiplication - we have got the cache utilization efficiency, memory bandwidth and parallelization efficiency metric quite easy. If projected to real tasks, where the matrix multiplication sizes are fixed, we easily predict the memory footprint and time necessary to complete the operation, so can evaluate the real performance (+-10%) on a GPU platform.

The paper is just the beginning of the road into the GPU performance analysis, as now we analyze only basic digits. We have not evaluated all simulation modes and have not elaborated all the parameters accessible in resulting log file, these are tasks for the further simulator evaluation. Also, we need more accurate "slicing" of simulation state in time, for gathering more precise information of algorithm behavior in real time. Additional analysis for cache memory utilization imbalance is also

required. Finally, we need to check the performance of algorithms which gives L2 miss rate more than 2% in order to check what we can get from DRAM channels load analysis.

8. References

[1] M. Khairy, A. Jain, T.M. Aamodt, T.G. Rogers. A Detailed Model for Contemporary GPU Memory Systems. 2019 IEEE International Symposium on Performance Analysis of Systems and Software (2019), p. 141-142.

[2] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, T. M. Aamodt. "Analyzing CUDA workloads using a detailed GPU simulator," 2009 IEEE International Symposium on Performance Analysis of Systems and Software (2009), pp. 163-174, doi: 10.1109/ISPASS.2009.4919648.

[3] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, C. R. Das. Anatomy of GPU Memory System for Multi-Application Execution. In Proc. of the 2015 International Symposium on Memory Systems (MEMSYS '15). ACM, NY, USA (2015), pp. 223–234. Doi: 10.1145/2818950.2818979

[4] M. A. Raihan, N. Goli, T. M. Aamodt. "Modeling Deep Learning Accelerator Enabled GPUs," 2019 IEEE International Symposium on Performance Analysis of Systems and Software (2019): pp. 79-92, doi: 10.1109/ISPASS.2019.00016.

[5] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, E. S. Quintana-Orti. "Evaluation and tuning of the Level 3 CUBLAS for graphics processors," 2008 IEEE International Symposium on Parallel and Distributed Processing (2008): pp. 1-8, doi: 10.1109/IPDPS.2008.4536485

[6] J. Kurzak, S. Tomov, J. Dongarra. "Autotuning GEMM Kernels for the Fermi GPU," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 11, pp. 2045-2057, Nov. 2012, doi: 10.1109/TPDS.2011.311.

[7] P. A. Ivanenko, A. Y. Doroshenko, K. A. Zhereb. TuningGenie: Auto-Tuning Framework Based on Rewriting Rules // in: 10th International Conference, ICTERI 2014, Kherson, Ukraine, June 9-12 (2014) Revised Selected Papers, Series: Communications in Computer and Information Science, Springer, CCIS Vol. 469 (2014): PP. 139-160. doi: 10.1007/978-3-319—13206-8_7

[8] Wu Kui, Truong Nghia, Yuksel Cem, Hoetzlein Rama. Fast Fluid Simulations with Sparse Volumes on the GPU. Eurographics/Computer Graphics Forum. Vol 37. May 2018. pp. 157-167. Doi: 10.1111/cgf.13350.

[9] Jain Akshay, Rogers Timothy. A Quantitative Evaluation of Contemporary GPU Simulation Methodology. ACM SIGMETRICS Performance Evaluation Review. Vol 46, June 2018. Pp. 103-105. Doi: 10.1145/3292040.3219658.

[10] M. Khairy, Z. Shen, T. M. Aamodt and T. G. Rogers, "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modelingm" 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), Valencia, Spain, 2020, pp. 473-486, doi: 10.1109/ISCA45697.2020.00047.

[11] L. Seiler, D. Carmean, E. Sprangle, etc.. Larrabee: A Many-Core x86 Architecture for Visual Computing. // IEEE Micro. 29. 2009, pp 10-21. 10.1109/MM.2009.9.