Minimizing Memory Contention in an APNG Encoder using a Grid of Processing Cells

Vivek Govindasamy, Emad Arasteh, and Rainer Dömer

CECS, University of California, Irvine
{vbgovind,emalekza,doemer}@uci.edu
https://www.cecs.uci.edu/

Abstract. Modern processors experience memory contention when the speed of their computational units exceeds the rate at which data can be accessed in memory. This phenomenon is well known as the memory bottleneck and is a great challenge in computer engineering. In order to mitigate the memory bottleneck in classic multi-core architectures, a scalable parallel computing platform called Grid of Processing Cells (GPC) has been proposed. To evaluate its effectiveness, we model the GPC using SystemC TLM-2.0, with a focus on memory contention. As an example, we parallelize an APNG encoder application and map it to the GPC and compare its performance to traditional shared memory processors. Our experimental results show improved execution times on the GPC due to a large decrease in memory contention.

Keywords: Memory Bottleneck · Grid of Processing Cells · SystemC TLM-2.0

1 Introduction

The increase in processor speeds over the past years has led to increased time spent in accessing the main memory to retrieve data. As many cores try to access the shared memory, this leads to contention and delays each core. The cores suffer from contention and their computations are halted due to sharing of the same main memory. This memory bottleneck applies to most modern CPUs which are usually shared memory processors (SMP).

To deal with slow memory access speeds, various solutions are being researched. The development of hierarchical caches is the main method to address this issue [1]. Another solution is the Berkeley RISC project, in which many complex instructions were removed because they were rarely used [2], and instead replaced with more CPU registers which are much faster to access than main memory [3][4].

In this paper, we model and evaluate SMPs and a scalable alternative called *Grid of Processing Cells (GPC)* where processors paired with local memories are arranged in a 2D array [5]. As a specific configuration, the GPC checkerboard architecture (Fig. 1) is aimed at addressing the memory bottleneck. The cores and memories are placed one after another, and each core has access to its

own and three neighbour memories, thereby increasing data availability. The checkerboard contains several variations of a main logical component which is termed as a cell. Each cell is designed with the idea that it represents a core and components that are local to that particular core.

In this work we model the GPC architecture in SystemC TLM-2.0 [6] and map an application to it [7].

1.1 Problem Definition

Our main contribution in this paper is modeling and demonstrating the improvement of the GPC against the classic SMP architectures in terms of execution time and time spent in main memory access contention when running an APNG encoder on the different architectures.

Fig. 1: 4-by-4 checkerboard GPC [5].

1.2 Background and Related Work

Over many years the general trend is that processors become faster as designers increase the clock rate but there is little increase in memory access speed [8]. This is known as the memory wall. To mitigate this problem, there has been a lot of focus on improving caches [1]. However, even the most advanced caches suffer from high miss rates if the cache size is too small or if associativity is increased too much [9]. Caches must also implement cache coherence protocols in the case of multi-core processors, and they consume significant space on the chip as well as power. Caches have given rise to Non-Uniform Memory Access (NUMA) where each core can access near memory faster than distant memory. With NUMA the time to maintain cache coherency is usually quite high [10] and leads to contention as the interconnect is shared for every core. There have

been other works related to addressing the memory wall, for example the Illusion system [11], which is similar to the GPC architecture. Both have a mapping step, but in the GPC the memories surrounding each core can be accessed only by neighbours and there is no NOC to facilitate communication between distant cores. This limitation makes the GPC truly scalable because there is no NOC complexity to grow, but puts a burden on application mapping which is restricted to only local communication.

Modification to the architecture itself is one method of reducing the contention between cores. In the checkerboard GPC contention is reduced drastically as separate buses are used between the cores. Architectures similar to the GPC have been proposed in the past, such as the Epiphany-V [12]. While similarities are present, such as the fact that both use a cache-less memory model, the GPC varies in a few aspects: i) The checkerboard architecture uses a different addressing map where each memory has a different address space, ii) The GPC has no operating system running on all of the cores, and iii) GPC uses simple multiplexers-based buses instead of a complex NOC.

Our main objective in this work is to reduce memory contention through software methods, which primarily lies in the mapping operation. We also confirm that memory contention is indeed a major reason for the memory wall, and provide experimental results showing that the GPC minimizes that.

2 Modeling of the Checkerboard GPC

The checkerboard model [5] consists of cores with local memory which can be accessed also by their neighbours. The memory size is small and the cores themselves only perform computation on small amounts of data at a time. The small memories are referred to as on-chip memories and are expected to be as fast as caches in a multi-core computer made of static random-access memory (SRAM). The off-chip memories are larger but slower, similar to dynamic random-access memory (DRAM).

Core Module - The core module is the computation component of a cell (Figure 2) and represents a complete processor core with in-order or out-of-order execution but without (or only 1st-level) cache. It contains a single socket connected to the core multiplexer. It is a SystemC module containing general arithmetic functions. The primary communication is the SystemC blocking transport interface (*b_transport*) [13] and event synchronization to prevent possible race conditions when interacting with other cores. Each core contains a main thread which performs the actual computation.

Memory Module - Each core has its own on-chip memory to store its data. This memory is assumed to be fast as SRAM, but small (up to 128 megabytes). The memory can only be accessed by the four neighbouring cores similar to a local scratchpad memory with explicitly managed address space. The off-chip

memories on the edges of the checkerboard are larger (512 megabytes) but have slower access (DRAM).

Core Demultiplexer Module - In order to communicate with the neighbours, an addressing scheme for the checkerboard is required. The global address space refers to the four off-chip memories on the outside and the local address space refers to the small memories near the cores. The core demultiplexer routes addresses to the individual memories. It contains one socket to communicate with the core and four sockets to connect to the adjacent memory multiplexers. The core demultiplexer forwards *b_transport* calls from the core after performing address translation.

Memory Multiplexer Module - The memory multiplexer is connected to a memory and to adjacent core demultiplexers. Its purpose is to forward *b_transport* calls from neighboring cores to its memory. The memory multiplexer permits only one access at a time and performs arbitration. This allows to observe memory contention. Algorithm 1 provides the algorithm of how the time spent waiting for memory access is computed [14].

Algorithm 1: Maintaining busy state in *b_transport* inside the core multiplexer (bus access arbitration with FCFS policy)

```
initialization: busy_until = 0;
busy = busy_until - current timestamp;
if busy < 0 then
    busy_until = current timestamp;
    busy = 0;
end
delay = multiplexer delay + busy;
d1 = delay;
socket->b_transport(transaction, delay);
d2 = delay;
memory delay = d2 - d1;
busy_until += memory delay ;
```

2.1 Modeling Interconnect Contention

To observe the contention in the traffic of memory transactions, we utilize timing delays in the SystemC TLM-2.0 blocking transport interface inside our interconnect. To ensure that only one transaction at a time uses each memory, the memory multiplexer stores its busy status in a state variable and delays competing transactions accordingly.

As listed in Algorithm 1, we store a timestamp marking the end of memory occupation in a variable `busy_until` which we initialize to zero. When a transaction arrives, we calculate the remaining time left until the memory becomes available again. If `busy` is negative, the transaction arrived at an idle time and `busy_until` is reset.

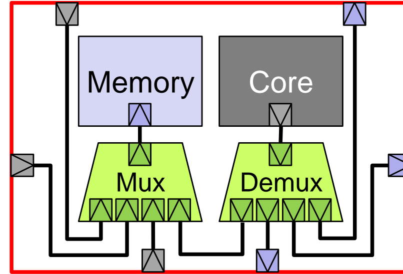

Fig. 2: Checkerboard GPC Cell.

Before forwarding the transaction to the memory (`b_transport`), we update the `delay` with the sum of the multiplexer latency and the busy delay. The transaction then processes in the memory. We observe the `memory_delay` by taking the time before and after the transaction and update the `busy_until` state variable accordingly.

In summary, our interconnect modeling accurately performs arbitration with first-come-first-serve policy and tracks the busy state of memory transactions. Aware of contention, our model enables accurate observation of any congestion in memory traffic.

3 Parallelized APNG Encoder Application

To test the performance of the checkerboard GPC, we need a suitable application which can be run in parallel. We choose an Animated Portable Network Graphics (APNG) encoder [15] which basically is a PNG encoder that concatenates generated PNG images with additional information such as the frame rate. PNG encoders have two main components which perform the actual image compression, the filters and the DEFLATE algorithm [16]. Filters are of five types (None, Sub, Up, Avg, Paeth) and are used to reduce pixel values. The reduced pixel values require less number of bits to transmit, providing some compression. The filtered values are sent in to the DEFLATE algorithm, which uses a combination of Lempel–Ziv–Storer–Szymanski (LZSS) and Huffman encoding to perform lossless compression. DEFLATE works better on values which are highly correlated to each other, which filtering provides [17].

Our SystemC TLM-2.0 APNG encoder consists of eight modules, namely the Color Splitter, Subtract Filter, Up Filter, Average Filter, Paeth Filter, Comparator, Compressor and APNG Encoder. The Color Splitter separates input data into individual color streams coming from the Stimulus of the encoder. The filters perform different mathematical computations which correlate pixel data, improving the compression provided by DEFLATE. The Comparator chooses the best filtered output to send to the Compressor. The Compressor uses DEFLATE to output a compressed row which is sent to the APNG Encoder module and the Monitor which writes the compressed data to a file. The APNG Encoder module generates additional information needed to create an APNG file which

is written in the Monitor module. Our model performs the encoding row wise, with parallel filters.

3.1 Backannotation of Delays

In order to evaluate performance, we need to reflect timing in the model. We estimate the computation delay of the major APNG functions by measuring their execution time on a reference platform. Since we are mainly interested in the relative timing of major blocks in the application, we simply run the APNG encoder on a computer (2.4 GHz CPU i5-1135G7) and measure the delays with the `gprof` Linux profiler. The observed delays are listed in Table I. We note that the filtering operations are most time-consuming in the encoder. Thus, we parallelize the filters in our model. We back-annotate the measured computation delays into the APNG SystemC model and scale them proportionally to the image size.

Table 1: APNG computation delays

Module Name	Total time	Time per frame	Time per pixel
Color Splitter	4s	0.133s	11ns
Subtract Filter	30s	1.000s	82ns
Up Filter	33s	1.100s	88ns
Average Filter	50s	1.667s	137ns
Paeth Filter	102s	3.400s	274ns
Comparator	8s	0.267s	21ns
Compressor	14s	0.467s	38ns
APNG Encoder	1s	0.033s	3ns

Every memory access by the cores results in a communication delay. In reality, not every memory access takes the same amount of time as some accesses will be to the cache and others to the main memory. However, we have not modeled caches in our SMP and single core models. Therefore, for fairness purposes we consider every memory access and multiplexer switch to be 10ns uniformly, regardless of on-chip or off-chip memory. This is still an effective measure of performance because the SMP models must perform main memory accesses frequently so that they can communicate the filtered rows to the other cores which use it, and caches are not of much use here.

Another important metric is the contention time, which is how much time each core spends waiting for access to the main memory, as described with Algorithm 1.

4 Mapping APNG on the Checkerboard

To evaluate the effectiveness of the checkerboard architecture we map the APNG encoder application on it. Since not all of the memories are available to each core, it is sometimes necessary to forward data through cores. Forwarding increases the communication time for each cell. To reduce the lost time incurred by forwarding, it is better to use data by an adjacent core.

4.1 Checkerboard Mappings

Two checkerboard mappings have been implemented, a simpler initial mapping where individual filtering on the three colors is performed on the same core, and

an improved mapping which splits the colors between cores to filter. In the initial mapping nine cores are involved in encoding with three cores forwarding data. The improved mapping has every core utilized in APNG encoding.

Initial Checkerboard Mapping The main idea behind the initial mapping is to avoid the use of excessive forwarding (Fig. 3 (a)). For instance, the Subtract filter receives the red, green, and blue values from the Color Splitter. Next, it performs the computation and sends both the filtered values and the unfiltered values to the Up filter. The Up filter performs its own computation and sends the filtered Sub, Up and unfiltered values to the Paeth filter. The Paeth filter then computes the Paeth filtered output, and sends it to the Comparator while also forwarding both outputs from the Up filter and Sub filter. The same process is performed in the Average filter route. At the Comparator the least sum filtered row is chosen and sent to the Compressor. The compressed output is forwarded through neighboring cells to the right and written to the output file using the monitor. It is also sent to the APNG Encoder module for APNG encoding and forwards it to the second monitor thread.

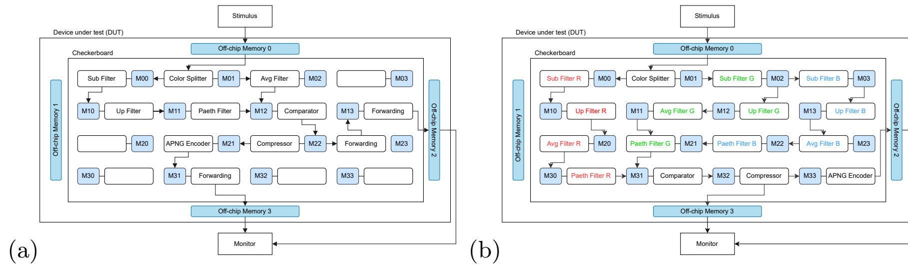

Fig. 3: (a) An initial checkerboard mapping which attempts to minimize communication between modules. (b) An improved checkerboard mapping which splits the computational load more evenly.

Improved Checkerboard Mapping While the initial mapping works, it could be further improved by splitting the filtering work of each core to three cores with each core filtering a different color component (Fig. 3 (b)). This is because some filters, such as the Paeth filter, take too much time to compute. Since there are sixteen cores available, it is possible to map every core to one module. This mapping is expected to be faster.

5 Experimental Results

We now compare five different SystemC models. The models are 1) Initial checkerboard (Fig. 3 (a)), 2) Improved checkerboard (Fig. 3 (b)), 3) Single core (Fig. 4

(a)), 4) 8 core SMP (Fig. 4 (b)), and 5) 16 core SMP (Fig. 4 (c)). These models are evaluated on the basis of their execution times and amount of contention.

5.1 Models for Comparison

For comparison, we implement a single core and also two SMP models in SystemC. These models use a single memory with a memory multiplexer connecting the cores to the memory. The single core model performs all computations in the same core (Fig. 4 (a)). The shared memory models work similar to the checkerboard architecture, using the same communication functions to transfer pixel data between cores, but have a greater amount of contention.

The SMP with 8 cores performs the filtering operation on the different color channels in the same core (Fig. 4 (b)). Doubling the number of cores provides a 16 core model in which the cores perform less work but communication is increased per unit time leading to higher contention (Fig. 4 (c)).

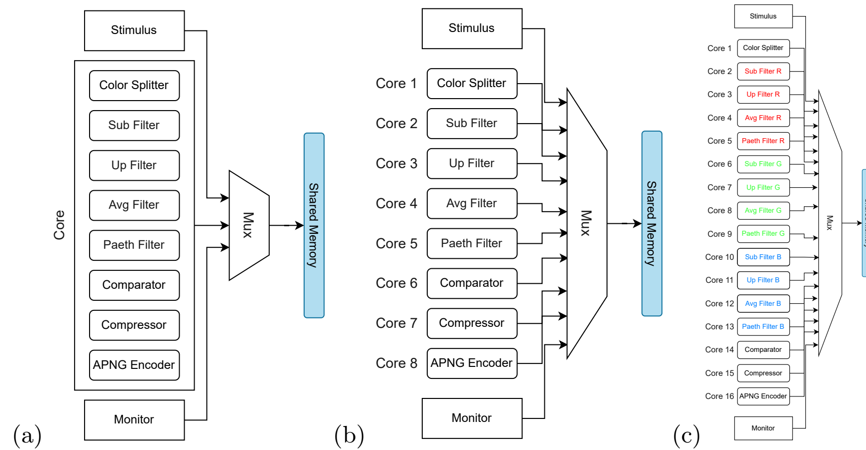

Fig. 4: Models for comparison, (a) single core architecture, (b) 8 core SMP where threads run in parallel, (c) 16 core SMP with shared memory.

5.2 Measurement of Delays

To measure delays, we create global variables of type *sc_time* called *Computation_Time*, *Read_Time*, *Write_Time* and *Contention_Time*. These variables provide an accurate value of how the time in each SystemC model is spent, and are present in every model. The *Computation_Time* (*Comp_Time*) is the summation of time spent by every core on the times listed in Table I.

Read_Time and *Write_Time* keep track of the total time spent by the model accessing the memory. Added together they reflect the *Memory_Access_Time* or the *MA_Time*.

Lastly, the *Contention_Time* (*Cont_Time*) is the total time spent by every core waiting to access the memory. This time can easily exceed the execution time of the model if there are a lot of cores attempting to access the memory at the same time, as each core will be waiting to access the memory and all of these times add up to the *Contention_Time*.

5.3 Simulated Time Results

With the three types of delays back-annotated, we obtain measurements on the five SystemC models. Table II shows the variation in model timings as the clock rate is gradually increased, thereby decreasing computation time per module and increasing the frequency of memory access requests. The five SystemC models are compared at different assumed processor clock rates. The clock rates have been chosen to start at 0.25 GHz and are doubled up to 8 GHz in our simulation.

Table 2: Table for simulated timing results

Model Name	Exec Time (Speedup)	Comp Time	MA Time	Cont Time
Clock rate of 0.25 GHz				
Single Core	714.617s (1x)	707.231s	4.619s	6.791s
SMP with 8 Cores	326.703s (1x)		18.518s	27.130s
SMP with 16 Cores	131.946s (1x)		18.525s	29.992s
Initial Checkerboard	334.866s (1x)		27.291s	1.388s
Improved Checkerboard	122.943s (1x)		28.296s	1.130s
Clock rate of 0.5 GHz				
Single Core	361.818s (1.98x)	353.615s	4.619s	6.791s
SMP with 8 Cores	166.704s (1.96x)		18.518s	61.124s
SMP with 16 Cores	81.454s (1.62x)		18.525s	64.398s
Initial Checkerboard	174.859s (1.92x)		27.291s	0.699s
Improved Checkerboard	70.008s (1.76x)		28.296s	2.676s
Clock rate of 1 GHz				
Single Core	185.419s (1.96x)	176.808s	4.619s	6.791s
SMP with 8 Cores	86.703s (1.92x)		18.518s	78.675s
SMP with 16 Cores	59.332s (1.38x)		18.525s	93.821s
Initial Checkerboard	94.855s (1.84x)		27.291s	0.354s
Improved Checkerboard	43.336s (1.63x)		28.296s	2.564s
Clock rate of 2 GHz				
Single Core	97.219s (1.91x)	88.404s	4.619s	6.791s
SMP with 8 Cores	48.423s (1.81x)		18.518s	88.403s
SMP with 16 Cores	45.426s (1.31x)		18.525s	101.670s
Initial Checkerboard	54.853s (1.73x)		27.291s	0.182s
Improved Checkerboard	30.001s (1.43x)		28.296s	2.712s
Clock rate of 4 GHz				
Single Core	53.119s (1.83x)	44.202s	4.619s	6.791s
SMP with 8 Cores	38.048s (1.26x)		18.518s	90.867s
SMP with 16 Cores	39.677s (1.13x)		18.525s	116.412s
Initial Checkerboard	34.852 (1.57x)		27.291s	0.096s
Improved Checkerboard	23.332 (1.31x)		28.296s	2.785s
Clock rate of 8 GHz				
Single Core	31.069s (1.71x)	22.101s	4.619s	6.791s
SMP with 8 Cores	38.047s (1.00x)		18.518s	91.792s
SMP with 16 Cores	38.308s (1.03x)		18.528s	119.683s
Initial Checkerboard	24.852s (1.41x)		27.291s	0.053s
Improved Checkerboard	19.999s (1.17x)		28.296s	2.619s

An important assumption made intentionally is that no cache memory exists in any model. The reasoning for this is that we want to observe memory contention directly, undisturbed by caching behaviour. In other words, for a fair comparison we model all memories as fast and thus do not need caches.

Table II shows the comparison of the three types of delays and the overall execution time. Time spent in computation linearly reduces, but has limited effect on the total execution time as the contention time goes up in every model, to varying degrees. The increase in contention time is the reason why execution times start to show less improvement.

Plotting the values from Table II provides insight on the change in execution time as the clock rate is doubled (Fig. 5). For the single core model, it is seen that the decrease in computation speed leads to great increase in speedup, until a certain point where diminishing returns are observed. The shared memory processors start off with low execution time, but they start to stagnate at around 4 GHz.

The *Memory_Access_Time* or *MA_Time* varies for each of these models even though the amount of pixels they process is the same. This is because simpler models, like the single core model, need to access the memory only two times (from the stimulus and to the monitor) when processing a row of pixel data. Other models, like the checkerboard, need to pass on the data between adjacent cores, which involves a lot more reading and writing from and to memories. The SMP models need to access the main memory frequently, as almost every core needs to access new data to continue data processing.

5.4 Observations and Comparison

The initial checkerboard mapping starts with an execution time similar to the 8 core SMP model, but quickly accelerates as the clock rate is increased. The improved checkerboard is as fast as the 16 core SMP at low clock rates, but is twice as fast when the clock rate reaches 8 GHz. At low clock rates, the limiting factor is the number of cores, and not the memory access contention, whereas at higher clock rates this trend is reversed.

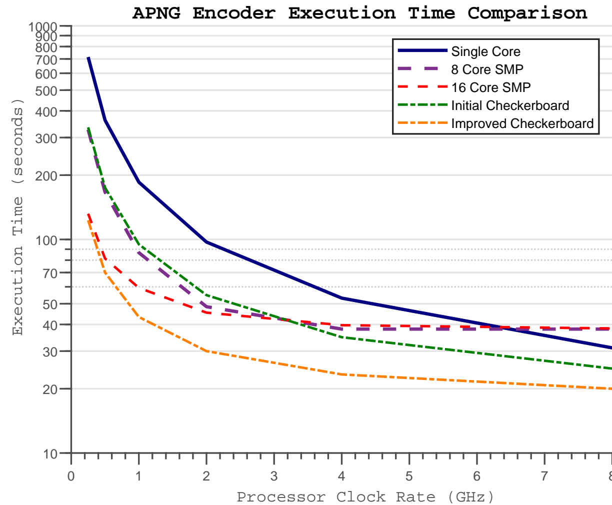

Fig. 5: APNG Encoder execution time scaling. Note that the graph y axis is in log scale.

Fig. 6 shows the increase in contention as the clock rate increases. This increases the rate at which memory is accessed by the cores so that they can process more data, which leads to a rise in contention time. For the single core model minimal contention exists because the cores are accessing the memory along with the stimulus and

monitor. The initial checkerboard mapping has a linear decrease in contention,

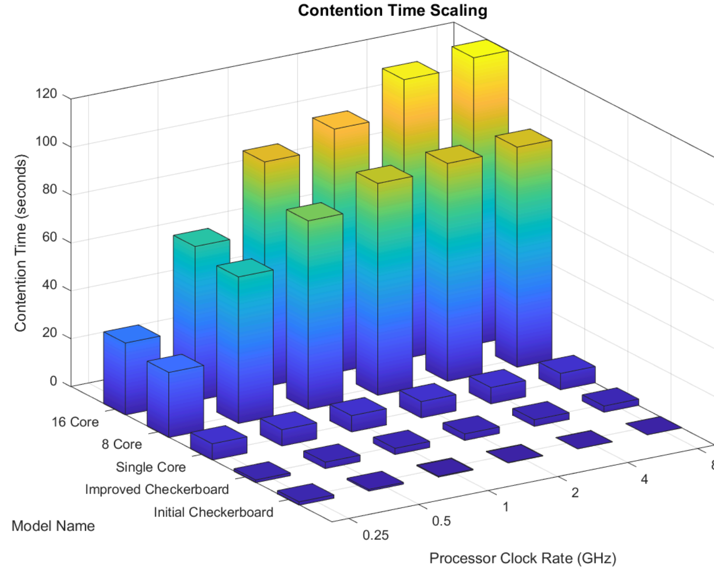

Fig. 6: Contention time scaling with increase in clock rate for the five SystemC models.

but the improved checkerboard suffers from a slight increase. The reasoning for the reduction in contention for the initial mapping is that when the clock rate increases the shorter memory accesses appear to come first, and this leads to a decrease in contention as shortest job first (SJF) reduces wait time. This does not seem to be the case for the SMP models however, as they have a noticeable increase in contention as the clock rate increases and the execution time starts to stagnate. Therefore the checkerboard architecture is a good alternative to shared memory processors as clock rate increases, its contention is much less.

6 Conclusion

The increase in processor speeds over the years has resulted in much faster computers but this trend has been hampered due to slower memory speed increases. The newly proposed checkerboard architecture is one possible way to mitigate the effects of slower memory access speeds, as shown by our experimental results.

In future work, we aim to provide more accurate comparisons by also including caches for our SMP models. Further, we would like to continue our modeling by lowering the level of abstraction of our cores to instruction set simulators [18] and mapping more applications to the GPC.

In the longer term, we plan to address the programmability of the GPC architecture so that applications can be mapped to it automatically by an advanced compiler.

References

1. A. J. Smith, "Cache memories," *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, 1982.
2. A. S. Tanenbaum, "Implications of structured programming for machine architecture," *Communications of the ACM*, vol. 21, no. 3, pp. 237–246, 1978.
3. D. A. Patterson, "Reduced instruction set computers," *Communications of the ACM*, vol. 28, no. 1, pp. 8–21, 1985.
4. J. Cocke and V. Markstein, "The evolution of RISC technology at IBM," *IBM Journal of research and development*, vol. 34, no. 1, pp. 4–11, 1990.
5. R. Dömer, "A Grid of Processing Cells (GPC) with Local Memories," Tech. Rep. CECS-TR-22-01, Apr. 2022.
6. T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemCTM.* Springer Science & Business Media, 2007.
7. V. Govindasamy, "Mapping of an APNG Encoder to the Grid of Processing Cells Architecture," Tech. Rep. CECS-TR-22-02, 2022.
8. S. A. McKee, "Reflections on the memory wall," in *Proceedings of the 1st conference on Computing frontiers*, 2004, p. 162.
9. D. A. Patterson and J. L. Hennessy, *Computer organization and design ARM edition: the hardware software interface.* Morgan kaufmann, 2016.
10. S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A Case for NUMA-aware Contention Management on Multicore Systems," in *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, 2011.
11. R. M. Radway, A. Bartolo, P. C. Jolly, Z. F. Khan, B. Q. Le, P. Tandon, T. F. Wu, Y. Xin, E. Vianello, P. Vivet *et al.*, "Illusion of large on-chip memory by networked computing chips for neural network inference," *Nature Electronics*, vol. 4, no. 1, pp. 71–80, 2021.
12. A. Olofsson, "Epiphany-v: A 1024 processor 64-bit risc system-on-chip," *arXiv preprint arXiv:1610.01832*, 2016.
13. "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, 2012.
14. E. M. Arasteh and R. Dömer, "Improving Parallelism in System Level Models by Assessing PDES Performance," in *2021 Forum on specification & Design Languages (FDL).* IEEE, 2021, pp. 01–07.
15. S. Parmenter, V. Vukićević, and A. Smith, "APNG Specification by Mozilla," accessed: 2021-08-12. [Online]. Available: https://wiki.mozilla.org/APNG_Specification
16. P. Deutsch and J.-L. Gailly, "Zlib compressed data format specification version 3.3," RFC 1950, May, Tech. Rep., 1996.
17. K. Sayood, *Chapter 9, Lossless compression handbook.* Elsevier, 2002.
18. V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Extensible and configurable RISC-V based virtual prototype," in *2018 Forum on Specification & Design Languages (FDL).* IEEE, 2018, pp. 5–16.