# Instruction-Level Modeling and Evaluation of a Cache-less Grid of Processing Cells

Vivek Govindasamy and Rainer Dömer
*CECS, University of California, Irvine,* Irvine, California, USA

2023 Forum on Specification & Design Languages (FDL) | 979-8-3503-0737-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/FDL59689.2023.10272195

***Abstract*—While processor speeds continue to show performance increases, memory access speeds remain significantly slower. One solution is to develop novel computer architectures, specifically designed to address the memory wall such as the cache-less Grid of Processing Cells (GPC). In this work, we model the GPC architecture using SystemC TLM-2.0 at the instruction-level with high accuracy, while retaining the characteristic high simulation speeds of transaction-level modeling. To compare the performance of our modeled GPC with existing architectures, we also model a single core and an 8-core SMP with optimized caches and run a bare-metal Canny application on the three architectures. We show promising performance evaluation results in architectures without caches.**

***Index Terms*—Embedded Systems, Memory Bottleneck, Virtual Prototyping, Grid of Processing Cells**

## I. Introduction

While processor speeds increase significantly every year, main memory access speed improvements remain significantly slower [1]. As many cores try to access the shared memory, it leads to contention and delays each core. This memory bottleneck applies to most modern CPUs which are usually shared memory processors (SMP). Almost all modern processors have local cache memories to store frequently used data [2] and avoid contention. Caches are becoming increasingly complex with multi-level hierarchies so that a miss is rare.

In this paper, we model and evaluate SMPs with highly optimized caches and a scalable alternative called Grid of Processing Cells (GPC) where cache-less processors paired with local memories are arranged in a 2D array [3]. Fig. 1 shows the GPC architecture in a 4×2 configuration. The gray boxes depict cores whereas the blue boxes show memories.

The GPC is a cache-less architecture with the primary goal of addressing the memory bottleneck. The checkerboard GPC has an alternating pattern of cores and memories, with four memories available per core. This design limits the maximum possible contention at a memory [4] and promises scalability [5]. This paper models the GPC accurately at the ISS level and provides extensive simulation results that show the cache-less performance benefits.

## II. Related Work

Computer architecture began with two types of initial architectures, the von-Neumann architecture [6] and Harvard architecture [7]. von-Neumann architectures share both data and instruction memory, while for Harvard architectures they are separated. The von-Neumann architectures are simpler but quickly experience contention between data and instructions. This is termed as the von-Neumann bottleneck [8]. The most widely used architecture has been the modified Harvard architecture for most general purpose computers [9], in which there exist separate caches for both instruction and data but the address space is shared. Caches have behaviour that is application-dependent and may not provide the same expected improvement in performance depending on how the programmer writes code [10]. Caches also implement cache coherence protocols in the case of multi-core processors [11] which creates complexity in their hardware design.

Most of all, caches consume significant space on the chip as well as power [12]. When the workload of a processor increases, caches do not provide sufficient performance increases. Non-Uniform Memory Access (NUMA) architectures are required where each core can access near memory faster than distant memory. Even with NUMA, the time to maintain cache coherency is usually high [13] and leads to contention as the interconnect is still shared for every core. There have been other works related to addressing the memory wall, such as the Illusion system [14], Tilera Tile64 processor [15], or the Epiphany-V [16]. Similarities exist between the GPC and these architectures, such as the fact that they use a cache-less memory model, but the GPC has a few differences. The GPC architecture uses an addressing map where each memory has a unique address space associated with it. GPC has no operating system and our current implementation is bare-metal. GPC uses interconnect multiplexers through which cores can only access neighbouring memories, and does not have a mesh network to support communication between distant cores. This reduces the complexity of the architecture, but requires the programmer to carefully map functions and pass data between the cores. In particular, the GPC is different from Network-on-Chip architectures [17] as communication between distant nodes is not supported in hardware. To run an application on the GPC, it must be manually mapped to the architecture, with the programmer being aware of which memories are used to communicate with other cores.

Our main objective in this work is to model and simulate these architectures with a high degree of precision, and to evaluate the practical viability of the GPC when compared to SMPs. We use an ISS [18] integrated in SystemC as a virtual prototype and customize its components so that the GPC architecture is accurately modeled at the ISS-level.

979-8-3503-0737-5/23/$31.00 ©2023 IEEE

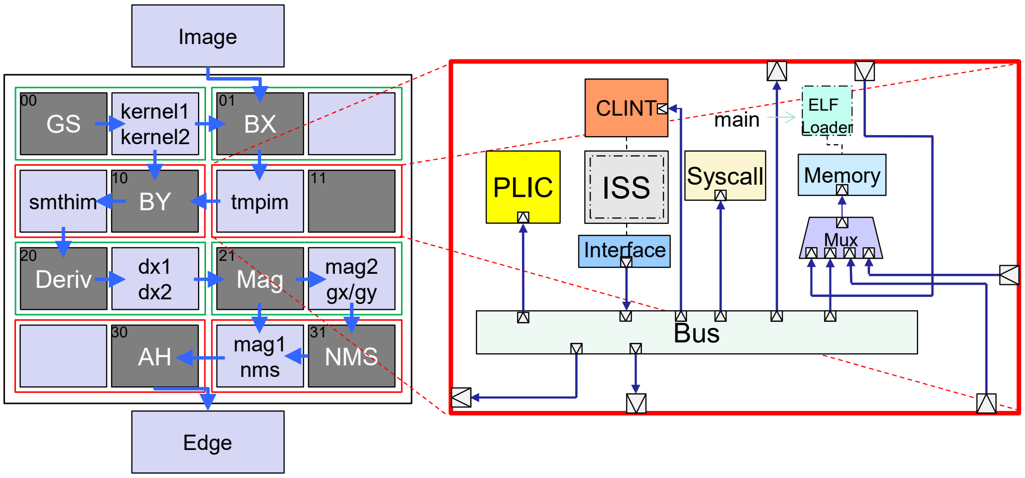

Fig. 1. A detailed diagram of the SystemC instruction-level model of the GPC architecture proposed in [3] with a mapped Canny application. The blue arrows on the left indicate inter-core data flow. On the right, blue arrrows represent TLM-2.0 blocking transport calls (*b_transport*).

## III. Problem Definition

In this work, we model and simulate the GPC, SMP, and a reference single core architecture, and run a Canny application on the three models. Our simulation is instruction accurate and takes into account complex caching behaviour for the classic models. We compare the three architectures in terms of execution time and other parameters as the clock rate increases and with varying memory configurations. Lastly, we discuss the practical viability of the GPC architecture.

### A. The Canny Application

Our Canny edge detector [19] application is a bare-metal implementation of the original source code [20]. We choose Canny as reference application because it is a diverse streaming application in embedded systems. Canny has seven functions, namely Gaussian Smooth (GS), Blur in X direction (BX), Blur in Y direction (BY), Derivative (Deriv), Magnitude (Mag), Non-Maximum Suppression (NMS), and Apply Hysteresis (AH). Each function requires partially computed image frames from the previous step. Some functions, such as BX, BY, Mag and NMS, use extensive floating-point operations[1]. Moreover, the Canny application exercises different memory access patterns in each block. For example, BX performs a horizontal filter and BY a vertical one. In contrast, AH performs a non-regular pixel traversal along the detected edges of the image by use of a recursive function FollowEdges with high stack usage. Other functions traverse multiple image arrays in parallel. So Canny exhibits diverse memory access patterns and is a well-suited benchmark for cache evaluation.

Fig. 1 shows the mapping of the Canny application on the GPC, with core functionality and the location of channels for communication. The testbench consists of Image and Edge devices which read the input and write the output images.

The Canny application is mapped identically on both the GPC and the SMP models. The models only differ in terms of where they read data from or write data to. The single core model has an advantage in this situation as it does not need to perform any memory-copy operations and does not need to wait for others to provide data. Each core's hardware thread is programmed to first copy the data from the neighbour's local memory (GPC) or shared memory (SMP), compute the function, and place the data back into another local or shared memory, respectively. Cores interrupt the others for new data as needed. This improves memory contention compared to using polling, but we also simulate polling and compare the two communication methods.

## IV. Architecture Modeling in SystemC TLM-2.0

Given the growing complexity of embedded systems, it is necessary to first model a design at higher abstraction levels to check whether it is suitable for implementation. For this, Electronic System Level (ESL) design and verification was introduced in the early 21st century [21]. An entire system can be modeled by using System-Level Design Languages (SLDLs), such as SystemC [22]. SystemC uses Transaction-Level Modeling (TLM) which allows communication between modules using a method call [23]. The original TLM-1 used channels to perform communication between modules. The channels were modeled as FIFOs or buses. However, to obtain a more accurate model of a real world design, it is essential to also include memories and explicit address-accurate transactions. For this purpose, SystemC TLM-2.0 was introduced where communication between modules takes place through explicit memories. Access to memories occurs through the call of a function, *b_transport*, whenever a read or write is required. Since SystemC TLM-2.0 provides memory-accurate modeling, we therefore choose it to model our GPC and other architectures at the instruction-level.

The GPC architecture consists of an array of cells. As shown on the right side of Fig. 1, each cell contains an

[1] This results in an unbalanced pipeline as the cores we use do not have any floating-point unit and use the SoftFloat library instead.

in-order 32-bit RISC-V ISS, a fast on-chip memory, a bus and other peripherals. Both instructions and data are stored in the local memory which is as fast as a cache (SRAM) but does not carry its coherence and update complexity. The RISC-V core accesses the memory in word sized *b_transport* function calls. The memory is primarily accessed by the core in the cell, but other cores may read and write from the memory through the use of the interconnect multiplexer for communication purposes. Instructions are loaded into the local memory through the use of an ELF loader SystemC module. This needs to be done for every cell, so a different cross-compiled executable file must be loaded which has the text segment shifted to match the address space of the GPC. A bus is present in each cell which maps addresses seen by the programmer to the physical addresses that the peripherals use.

Each memory has a unique address space associated with it. This changes based on the size of the local memories. To access a specific memory which is a neighbour to the core, the programmer can create a pointer to the address and read or write to the address of the pointer. Both the bus and interconnect multiplexer are modeled to show contention [24]. So memory accesses slow down if two cores attempt to access a memory at the same time. Other peripherals attached to the bus include a Core Local Interrupt controller (CLINT), system calls interface, and a Programmable Logic Interrupt controller (PLIC). These peripherals are present in every cell so that the GPC is fully scalable to any number of cores. A cell can communicate with its neighbours using polling or generating interrupts. A GPC core interrupts all of its neighbouring cells when it generates an interrupt.

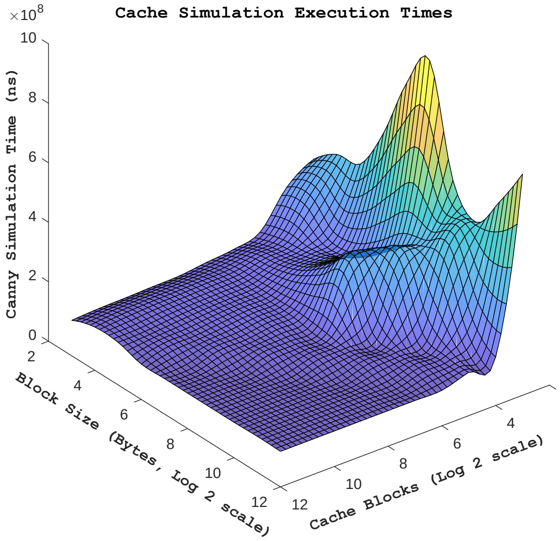

Fig. 2. Cache simulation results validate the functionality of the cache.

For a fair comparison between the GPC and the SMP, it is necessary to model SMP caches, as the main novelty of the GPC is the idea that it is *cache-less* and does not have the complexity that caches bring. We model a highly configurable write-through[2] cache module for the SMP and single core architectures. We use the least recently used (LRU) cache replacement with fully associative placement policy[3].

To confirm the correct functionality of our cache, we simulate the last four stages of the Canny application on a single core model with hundred different cache configurations of block sizes and number of cache blocks. Running the simulation (Fig. 2) shows that larger cache sizes uniformly perform better than smaller ones, which perform poorly because of high miss rates. While there is some variation in terms of block sizes compared to the number of blocks, it is not too significant, and only matters when the size of the cache is small.

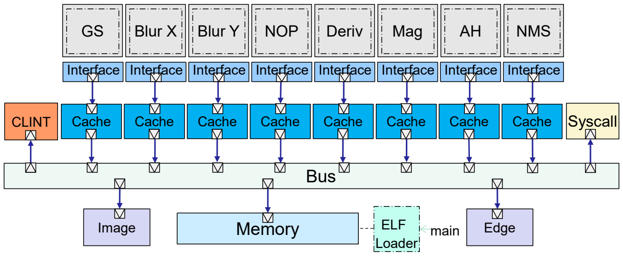

Fig. 3. Modeled shared memory processor. The blue lines represent SystemC blocking transport calls (*b_transport*).

Using the modeled caches, we create an SMP model (Fig. 3) which has eight cores and a single shared memory. Since the SMP model has multiple caches, it is necessary that the data is coherent between every single cache. Maintaining cache coherence can be time consuming and can cause main memory accesses, leading to contention. Our modeled cache is write-through, and we update every cache when a write happens. This process is assumed to take no time beyond the cache access itself and provides a fair comparison between the GPC and the SMP architectures. The SMP architecture only experiences contention on the bus. Similar to the SMP model we also create a single core model which performs the entire Canny computation on a single core. The model closely matches the SMP architecture shown in Fig. 3 but has only one core.

## V. Model Evaluation

Before evaluating the models, we first provide details about our experimental setup. To provide maximum accuracy for the simulation, we set the SystemC tlm_global_quantum to 0. The Image and Edge modules, which are the test bench, have both zero delays associated with them. They use a SystemC event to synchronize reading or writing, which represents an interrupt.

Table I shows the parameters which will be modified when running the simulation. The main memory access time is set to 50 ns for the single core and SMP models. The GPC does not use any main memory. The caches and the on-chip memory

[2] We choose write-through caches because they are simpler to model with cache coherency and bus contention. Other caches are future work.

[3] While a fully associative cache may not be realistic, it minimizes the miss rate and thus models best-possible SMP performance (which GPC still beats).

are varied in access time from 10 ns, to 5 ns and lastly 2 ns. CPU cycle time is approximately halved from 20 ns five times to 1 ns. This varies the clock rate from 50 MHz to 1 GHz. Three different cache sizes are used for the purposes of the simulation, with different configurations. This gives us an idea of the performance of these architectures across a wide span of configurations.

TABLE I
TABLE FOR EXPERIMENTAL PARAMETERS

| Model Name | Memory Access Time (ns) | Cycle Time (ns) | Cache Size (Blocks × Byte blocks) |
|---|---|---|---|
| **Single Core** | Main memory = 50<br>Cache = 10/5/2<br>Cache read hit = 10/5/2<br>Cache write hit = 10/5/2+50<br>Cache read/write miss = 10/5/2+50 | 20<br>10<br>5<br>2<br>1 | 8 kB = 128 × 64<br>64 kB = 256 × 256<br>512 kB = 2048 × 256 |
| **8 Core SMP** | Main memory = 50<br>Cache access = 10/5/2<br>Cache read hit = 10/5/2<br>Cache write hit = 10/5/2+50<br>Cache read/write miss = 10/5/2+50<br>Cache update = 0 | 20<br>10<br>5<br>2<br>1 | 8 kB = 128 × 64<br>64 kB = 256 × 256<br>512 kB = 2048 × 256 |
| **4 × 2 GPC** | On-chip memory = 10<br>On-chip memory = 5<br>On-chip memory = 2 | 20<br>10<br>5<br>2<br>1 | No Cache |

We configure Canny to send four frames through the pipeline. We use small images of 100x56 pixels in resolution and 4x larger ones (200x112 pixels) for cache evaluation.

## VI. EXPERIMENTAL RESULTS

We now present our simulation results which show the superior performance of the cache-less GPC over the classic architectures.

### *A. Initial Validation*

To confirm the functionality of our model, we first simulate the 4×2 GPC and 8-core SMP with no contention, without caches, and 0 ns memory access time. The CPU cycle time is set to 10 ns. We compile the code for both architectures with flags -march=rv32ima and -mabi=ilp32, with no additional compiler optimizations. The SMP completes this execution in 0.656 seconds and the GPC in 0.658 seconds. Here, the GPC is slightly slower since its startup code is more complex. This verifies that both models and executables are correctly and fairly implemented. Despite the small image resolution, the single core, SMP and GPC models take around 20 minutes to simulate due to the instruction-accuracy.

### *B. Variation in Cache Sizes with 2 ns Hit Time*

In our first experiment, we set the cache hit time (single core and SMP) and on-chip memory access time (GPC) to 2 ns. We use the same flags mentioned above to compile our code, but also add the -O3 flag to enable aggressive optimizations. This is the fastest possible memory access time that we have simulated. Now we compare the effect that the cache configurations have on the execution times as the clock speed rises. Initially the clock cycle time is 20 ns (50 MHz). This value is stepwise improved by a factor of 20x and the final simulation has a cycle time of 1 ns (1 GHz).

TABLE II
MEASURED EXECUTION TIMES (S) AND CONTENTION TIMES AT 2 NS SRAM ACCESS TIME (100X56 RESOLUTION IMAGES)

| Single Core | | | SMP | | | GPC |
|---|---|---|---|---|---|---|
| **8 kB** | **64 kB** | **512 kB** | **8 kB** | **64 kB** | **512 kB** | **none** |
| **Cycle Time of 20 ns** | | | | | | |
| 4.7696 | 3.7302 | 3.7137 | 3.2720<br>(74 ms) | 1.3546<br>(89 ms) | 1.3358<br>(52.9 ms) | 1.2103<br>(14.2 us) |
| **Cycle Time of 10 ns** | | | | | | |
| 3.1891 | 2.1497 | 2.1332 | 2.3551<br>(144 ms) | 0.8189<br>(199 ms) | 0.7891<br>(116 ms) | 0.6457<br>(5.85 us) |
| **Cycle Time of 5 ns** | | | | | | |
| 2.3989 | 1.3594 | 1.3429 | 1.9304<br>(275 ms) | 0.5799<br>(368 ms) | 0.5437<br>(250 ms) | 0.3634<br>(19.7 us) |
| **Cycle Time of 2.5 ns** | | | | | | |
| 1.9247 | 0.8853 | 0.8688 | 1.7102<br>(457 ms) | 0.4637<br>(564 ms) | 0.4201<br>(410 ms) | 0.1940<br>(41.1 us) |
| **Cycle Time of 1 ns** | | | | | | |
| 1.7667 | 0.7272 | 0.7107 | 1.6518<br>(561 ms) | 0.4415<br>(706 ms) | 0.3905<br>(509 ms) | 0.1376<br>(130 us) |

Table II shows the variation in simulated times as the cycle time reduces for the different models with small images. The execution time is noted above the contention time which denotes how much total time was spent waiting to access the memory by every core. This value can exceed the execution time of the model as the time spent by every core is added up. For the GPC, the contention time of every multiplexer in the system is noted and summed up. This value is always zero for the single core model and is not shown. The data is visualized in Fig. 4 for our two image sizes. We observe that the single core and SMP models perform better with a 64 kB or 512 kB cache over a 8 kB cache. The 8 kB cache is too small resulting in a high number of capacity and conflict misses. The contention time reduces, thereby improving the execution time. Increasing the cache size to 512 kB from 64 kB only results in slight performance increases, as the cache becomes saturated. The larger image size slows down the single core model considerably because the amount of data processing increases. More cores to increase parallelism is beneficial for both the SMP and GPC models, but the GPC performs best.

The GPC and SMP models perform roughly the same at high CPU cycle times, while being significantly better than the single core model. At high clock rates/low CPU cycle times the performance of the SMP model starts to stagnate (around 5 ns cycle time), when compared with the single core model and the GPC. This can be attributed to the increased contention between the cores as they become faster. The average hit rate of the modeled caches is listed in Table III. The hit rates are quite high, as Canny is an application which uses large arrays, which are stored next to each other in the memory. As the image resolution is increased, the cache hit rates vary, but the overall hit rate goes down, especially for the smaller caches which cannot hold the entire images. Despite the very high hit rate, the performance is slower than the GPC, because a miss and subsequent main memory access is expensive and can lead to contention. Overall, these results show that the GPC is the faster architecture at high clock rates, and more importantly scales better due to reduced memory contention.

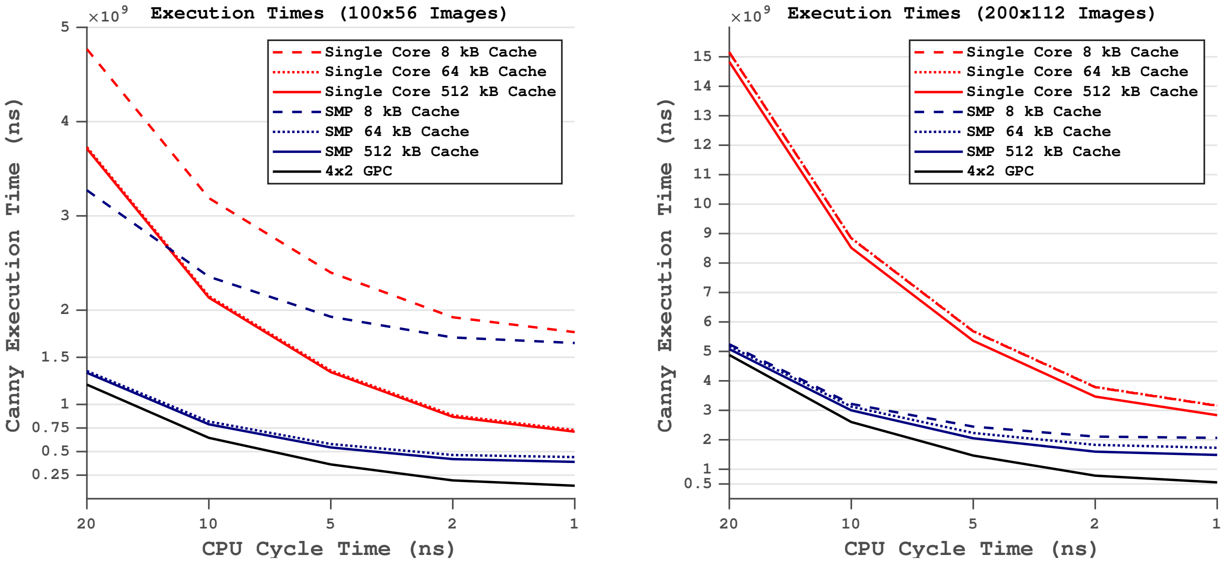

Fig. 4. Comparing the execution times when the SRAM (fast) memory has a 2 ns access delay for two different image sizes.

TABLE III
Cache Hit Rates for SMP Models

| **Core Functionality** | **100x56 Resolution Image** | | | **200x112 Resolution Image** | | |
|---|---|---|---|---|---|---|
| | 8 kB Cache | 64 kB Cache | 512 kB Cache | 8 kB Cache | 64 kB Cache | 512 kB Cache |
| Core 0: Gaussian Kernel | 98.87% | 99.84% | 99.85% | 98.74% (-0.13%) | 99.84% (0.00%) | 99.84% (-0.01%) |
| Core 1: Blur X | 99.98% | 99.99% | 99.99% | 99.93% (-0.05%) | 99.98% (-0.01%) | 99.99% (0.00%) |
| Core 2: Blur Y | 99.88% | 99.99% | 99.99% | 99.96% (+0.08%) | 99.89% (-0.10%) | 99.99% (0.00%) |
| Core 3: NOP | 99.70% | 87.06% | 87.06% | 86.44% (-13.26%) | 92.52% (+5.46%) | 92.52% (+5.46%) |
| Core 4: Derivative | 98.02% | 99.91% | 99.98% | 94.55% (-3.47%) | 99.78% (-0.13%) | 99.98% (0.00%) |
| Core 5: Magnitude | 99.94% | 99.98% | 99.99% | 99.90% (-0.04%) | 99.97% (-0.01%) | 99.99% (0.00%) |
| Core 6: Apply Hysteresis | 99.97% | 99.79% | 99.97% | 99.08% (-0.89%) | 99.82% (+0.03%) | 99.98% (+0.01%) |
| Core 7: Non-Maximum Suppression | 98.87% | 99.99% | 99.99% | 99.95% (+1.08%) | 99.98% (-0.01%) | 99.99% (0.00%) |

### *C. Variation in SRAM Access Time and CPU Cycle Time*

For our next set of simulations, we set the cache size to 512 kB, which executes the fastest as shown in Fig. 4. The simulations are run on the different set of parameters listed in Table I. Our results are graphically shown in Fig. 5. In these results the x-axis denotes the speed at which the cache or on-chip memory can be accessed. The y-axis is the cycle time of the CPU. The z-axis is the total time it takes for the Canny simulation to complete. We show this plot for the small images on the left and the larger images on the right. The inference that we obtain from Fig. 5. is that as the clock speed rises, execution times fall drastically at first. However, for the change of cycle time from 5 ns to 1 ns the execution time of the SMP model does not change much. We observe that in this instance, it is more important to improve the cache access speed.

The SMP and GPC have similar execution times when the cycle time and cache access times are higher, and even when only one of the two is high. We discuss these results for the small images first. When both of these values start to decline, the GPC becomes significantly faster, which we observe when the clock cycle time is 1 ns and memory access time is 2 ns. In this situation, the GPC simulates in 0.1376 seconds compared to the SMP which takes 0.3905 seconds. The GPC has an execution speed advantage of around 3x in this situation. This happens because as the cores become faster relative to the memory, they request data and instructions from the main memory in shorter time, which increases bus contention and thus slows down the execution time. The GPC also experiences this issue, but since the memories are shared by few cores at a time the contention is far less, even without caches. This trend is weaker at low cycle times and cache hit times, such as at 20 ns cycle time and 10 ns hit time. In this situation, the SMP simulates in 1.3358 seconds while the GPC simulates in 1.2103 seconds. The SMP is only 10% slower for this set of parameters. There are other configurations in which the performance of the SMP and GPC are very close to each other. When using the larger images, we see similar results with the GPC performing about the same as the SMP at high CPU cycle times and low memory access speeds. We conclude from Fig. 5 that as the clock rate and SRAM access speeds increase, the contention also increases, which results in the SMP model having slower execution times. The slower execution speeds due to contention can be minimized with GPC-based architectures.

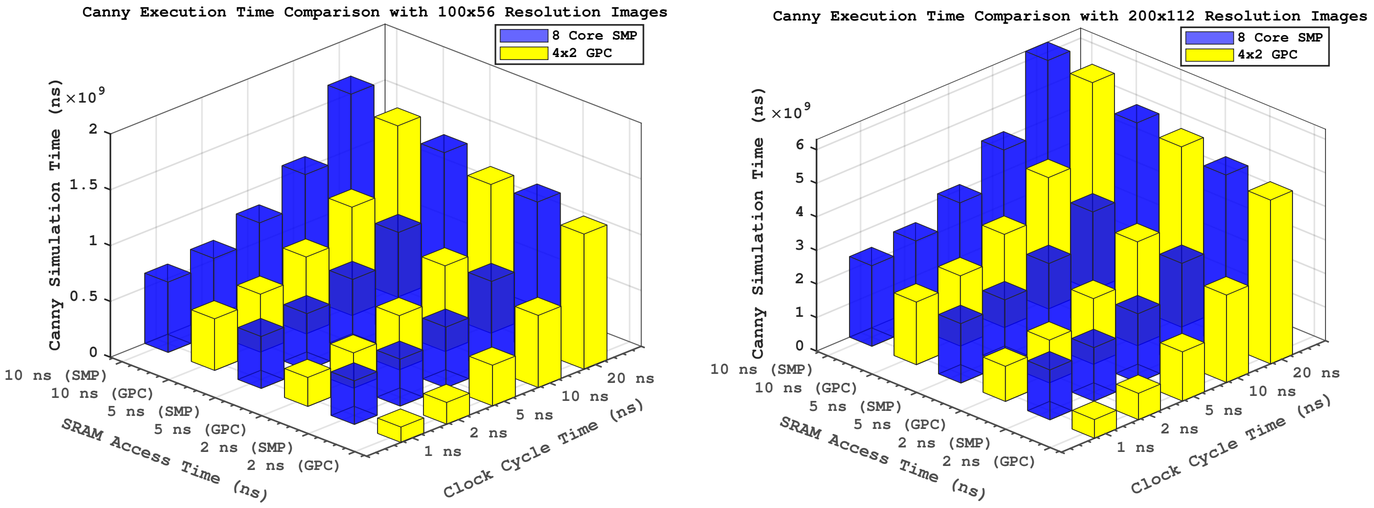

Fig. 5. Detailed performance comparison between the three models when the cache size is set to 512 kB for two different image sizes.

TABLE IV
EXECUTION SPEED COMPARISON OF INTER-CORE COMMUNICATION METHODS WITH 1 NS CPU CYCLE TIME AND 2 NS SRAM ACCESS TIME (100X56 IMAGE RESOLUTION)

| Communication Method | SMP 512 kB Cache | | GPC | | Separate Instruction Memory GPC | |
|---|---|---|---|---|---|---|
| | Exec. Time | Cont. Time | Exec. Time | Cont. Time | Exec. Time | Cont. Time |
| Interrupts | 390446572 ns | 509774095 ns | 137614168 ns | 116770 ns | 137556163 ns | 5002 ns |
| Polling | 393557136 ns | 516821654 ns | 150965762 ns | 38479247 ns | 137697451 ns | 45876 ns |
| Polling w/ 10 ms Timer | 417862965 ns | 547786437 ns | 172559562 ns | 158117 ns | 172556324 ns | 7 ns |
| Polling w/ 1 ms Timer | 395031155 ns | 529033013 ns | 140560011 ns | 145459 ns | 140557310 ns | 12 ns |
| Polling w/ 0.1 ms Timer | 431762008 ns | 781635331 ns | 138530176 ns | 207582 ns | 138428065 ns | 4747 ns |

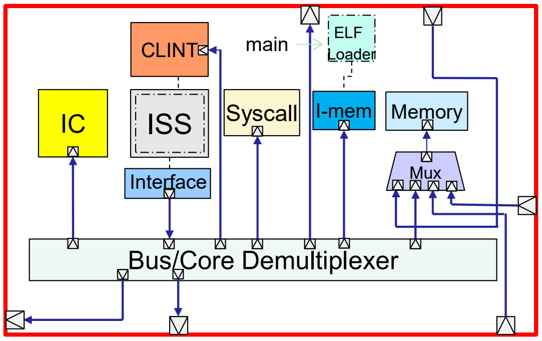

Fig. 6. The GPC cell with a separate memory for instructions. The memory multiplexer experiences less contention in this model.

### D. Adding a Separate Memory for Instructions

Polling is a simpler method to communicate between threads or cores. But the disadvantage of it is that the memory is frequently accessed to check whether a variable has changed. For the SMP model, this is not an issue, because if the data does not change, then the memory access only goes to the local cache and not to the main memory. The increase in contention due to polling is minimal, but multiple clock cycles are spent checking if the variable has changed in the local cache. This causes an increase in power consumption, but the execution speed remains similar. Polling can have a significant impact for the GPC architecture however. Since the instructions of every core are stored in its own memory, when a neighbouring core attempts to access the same memory to check whether the polling variable has changed, it leads to increased contention in the interconnect memory multiplexer. To resolve the increase in contention due to polling we have also modeled a variant of the GPC in which the instruction memory is directly connected to the bus and the memory used for communication is connected to the interconnect multiplexer (Fig. 6). Both of these memories will become smaller in size. The performance improvement is measured in Table IV. The disadvantage of separating the instruction memory is that the code segment and communication arrays must be smaller to fit inside the reduced memory sizes.

### E. Comparing Different Communication Methods

The GPC is designed to communicate using interrupts, as interrupts reduce the power consumption when the core goes to sleep. We also compare the communication when the cores communicate using polling and when they communicate using a timer-generated interrupt. Special addresses are present in the on-chip memory on the GPC which, when written to, generate an interrupt waking up its neighbours. A similar mechanism is present for the SMP model in which only a specific core wakes up when the interrupt is generated by writing to a designated memory address. In other words, in the SMP model a core can

TABLE V

Table for instruction count comparison with 1 ns CPU cycle time and 2 ns SRAM access time (100x56 image resolution)

| **Communication Method: Interrupts** | | | | | | |
|---|---|---|---|---|---|---|
| **Core Functionality** | **SMP 512 kB Cache** | | **GPC** | | **Separate Instruction Memory GPC** | |
| | Instructions | WFI | Instructions | WFI | Instructions | WFI |
| Core 0: Gaussian Kernel | 0.1 M | 2 | 0.1 M | 0 | 0.1 M | 0 |
| Core 1: Blur X | 19.0 M | 8 | 18.5 M | 9 | 18.5 M | 9 |
| Core 2: Blur Y | 35.5 M | 1 | 32.8 M | 3 | 32.8 M | 3 |
| Core 3: NOP | ≈ 0.0 M | 0 | ≈ 0.0 M | 98 | ≈ 0.0 M | 101 |
| Core 4: Derivative | 1.4 M | 20 | 1.4 M | 16 | 1.3 M | 16 |
| Core 5: Magnitude | 21.7 M | 14 | 21.4 M | 36 | 21.4 M | 36 |
| Core 6: Apply Hysteresis | 2.2 M | 4 | 2.2 M | 24 | 2.2 M | 24 |
| Core 7: Non-Maximum Suppression | 26.4 M | 1 | 26.3 M | 1 | 26.3 M | 1 |
| **Communication Method: Polling** | | | | | | |
| **Core Functionality** | **SMP 512 kB Cache** | | **GPC** | | **Separate Instruction Memory GPC** | |
| | Instructions | WFI | Instructions | WFI | Instructions | WFI |
| Core 0: Gaussian Kernel | 0.1 M | 0 | 0.1 M | 0 | 0.1 M | 0 |
| Core 1: Blur X | 50.8 M | 0 | 35.1 M | 0 | 32.2 M | 0 |
| Core 2: Blur Y | 38.4 M | 0 | 34.3 M | 0 | 34.2 M | 0 |
| Core 3: NOP | ≈ 0.0 M | 0 | ≈ 0.0 M | 0 | ≈ 0.0 M | 0 |
| Core 4: Derivative | 70.9 M | 0 | 29.2 M | 0 | 27.6 M | 0 |
| Core 5: Magnitude | 57.7 M | 0 | 36.4 M | 0 | 34.2 M | 0 |
| Core 6: Apply Hysteresis | 67.1 M | 0 | 29.3 M | 0 | 27.7 M | 0 |
| Core 7: Non-Maximum Suppression | 41.8 M | 0 | 33.6 M | 0 | 33.4 M | 0 |
| **Communication Method: Polling with a 0.1 ms Timer** | | | | | | |
| **Core Functionality** | **SMP 512 kB Cache** | | **GPC** | | **Separate Instruction Memory GPC** | |
| | Instructions | WFI | Instructions | WFI | Instructions | WFI |
| Core 0: Gaussian Kernel | 0.4 M | 4277 | 0.2 M | 1383 | 0.2 M | 1382 |
| Core 1: Blur X | 19.4 M | 1625 | 18.5 M | 648 | 18.5 M | 648 |
| Core 2: Blur Y | 35.8 M | 152 | 32.9 M | 73 | 32.9 M | 73 |
| Core 3: NOP | 0.4 M | 4313 | 0.1 M | 1385 | 0.1 M | 1384 |
| Core 4: Derivative | 1.8 M | 3816 | 1.4 M | 1322 | 1.4 M | 1321 |
| Core 5: Magnitude | 22.1 M | 1888 | 21.5 M | 628 | 21.5 M | 628 |
| Core 6: Apply Hysteresis | 2.6 M | 3559 | 2.3 M | 1285 | 2.3 M | 1284 |
| Core 7: Non-Maximum Suppression | 26.8 M | 848 | 26.4 M | 359 | 26.4 M | 359 |

interrupt any other specific core. The cores can communicate using polling as well. The last method of communication is timer generated interrupts (polling at a regular interval) where the core goes to sleep when it is waiting for data, but wakes up periodically due to a timer-generated interrupt to check for the availability of new data. We compare the execution times of these three communication methods for the SMP, GPC and Separate-Instruction-Memory GPC. Our results for small images are tabulated in Tables IV and V.

From Table IV, we observe that the execution times for interrupts and polling in the SMP model are very close to each other. When using timer-based interrupts we observe that the fastest execution speed is achieved with an interrupt frequency of 1 ms. For the GPC model polling slows down the execution from 0.137s when using interrupts to 0.15s, a slow down of roughly 9.4%. When using a separate instruction memory in the GPC, the slowdown is very small, which implies that creating two separate memories per cell can be beneficial if the user decides to communicate using polling. Polling at a regular interval does not improve execution speeds beyond only interrupts/polling, but we observe that the memory access contention is reduced in some cases. The fastest execution times are observed when using interrupts only, with no polling in all of the three models.

Table V shows the amount of instructions executed per core in the three models. When using interrupt-based communication, the models nearly match in terms of instructions executed. When the communication method is polling, the SMP model performs significantly more instructions depending on the core. For example, Core 4, responsible for the Derivative function, performs 1.37 million instructions when using interrupts to communicate compared to 70.9 million instructions when using polling, an increase of 50x. This leads to much higher power consumption, even though the execution time remains similar for the SMP model. The GPC models execute more instructions, but the increase is not as high as for the SMP model. The reason for this is that the multiplexer contention increases the delay of memory access requests, which leads to fewer memory accesses to check whether the variable has changed. When polling, the models with the fastest execution time also have cores which execute less instructions since they do not poll for new data as much as a core in a model which takes longer to execute.

The number of wait-for-interrupt (WFI) instructions has also been noted in Table V. We see that the number of WFI instructions is high when the communication method is timer-based interrupts. When handling an interrupt, the program context needs to be stored on the stack, and this results in memory accesses. These memory accesses can lead to increased contention in the SMP model as we use write-through caches, and slow down the execution. This issue also exists in the GPC models, but the lesser chance for contention and faster memory writes result in the execution time not dropping as much. Some cores, such as core 0 and core 4, which have little work to do, execute WFI instructions proportional to the execution time of the Canny application.

The best communication method for the GPC is using interrupts, as it has the best execution times, with the least number of instructions being executed. Ideally, a GPC implemented on hardware would communicate using solely using interrupts. However, in a real hardware implementation, it may not be possible to communicate solely using interrupts, so timer-based interrupts may be the better option.

### F. Cache vs On-chip memory

The GPC stores both instructions and data in the same on-chip memory. This results in a situation where the on-chip memory is very similar to a main memory but has very fast access times. The practical viability of such a memory must also be evaluated. When compiling the Canny code for the SMP architecture, the text segment was 92 kB. For the GPC it varied from between 7 kB to 17 kB. The size of the executable program/instructions is not an issue for the GPC as the code becomes smaller when less functions and variables are present. Memory is also required for the large arrays created, which is dependent on the processed data sizes. Since the GPC does not have a large shared memory in the current configuration, the data size that it can process is limited compared to the SMP and single core architectures. To alleviate this, the four edges of the GPC can be connected to larger off-chip memories which can increase the processing data size, but accessing the off-chip memory will be slower and can cause a performance bottleneck. In future work we plan to record the memory utilization of each core precisely.

Another advantage of the on-chip memories is that they do not require any replacement policies or coherency protocols. Every core sees the same data, thereby providing a simpler design from a hardware point of view. This reduces the hardware logic requirements of the memory components, thereby reducing the surface area on the chip. We anticipate that the GPC architecture will be cheaper to synthesize and have less die area.

## VII. Conclusion

We have modeled the GPC architecture at the instruction-set level with high accuracy and have shown in extensive simulation and evaluation that the cache-less GPC has significant performance advantages over traditional single and multi-core architectures. In the future we would like to continue to try a more diverse range of applications. Complex data-intensive applications which use a large number cores will be ideal for the GPC architecture to excel at. The more challenging and long term objective is to create an advanced compiler which can automatically take a program and find an optimized mapping for the given GPC configuration, as well as create executable files for each core.

*Acknowledgment:* The authors thank the anonymous reviewers for their valuable suggestions to improve this work.

## References

[1] S. A. McKee, "Reflections on the memory wall," in *Proceedings of the 1st conference on Computing frontiers*, 2004, p. 162.

[2] A. J. Smith, "Cache memories," *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, 1982.

[3] R. Dömer, "A Grid of Processing Cells (GPC) with Local Memories," CECS, UCI, Tech. Rep. CECS-TR-22-01, Apr. 2022.

[4] Vivek Govindasamy, E. Arasteh, and R. Dömer, "Minimizing Memory Contention in an APNG Encoder using a Grid of Processing Cells," in *IESS*, Lippstadt, Germany, Nov. 2022.

[5] Yutong Wang, A. Daroui, and R. Dömer, "Demonstrating Scalability of the Checkerboard GPC with SystemC TLM-2.0," in *IESS*, Lippstadt, Germany, Nov. 2022.

[6] J. Von Neumann, "First Draft of a Report on the EDVAC," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993.

[7] W. Stallings, *Computer organization and architecture: designing for performance*. Pearson Education India, 2003.

[8] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[9] G. Frantz, "Digital signal processor trends," *IEEE micro*, vol. 20, no. 6, pp. 52–59, 2000.

[10] D. A. Patterson and J. L. Hennessy, *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann, 2016.

[11] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis lectures on computer architecture*, vol. 6, no. 3, pp. 1–212, 2011.

[12] U. Ko, P. T. Balsara, and A. K. Nanda, "Energy optimization of multilevel cache architectures for risc and cisc processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 2, pp. 299–308, 1998.

[13] 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.

[14] 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.

[15] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown *et al.*, "Tile64-processor: A 64-core soc with mesh interconnect," in *2008 IEEE International Solid-State Circuits Conference-Digest of Technical Papers*. IEEE, 2008, pp. 88–598.

[16] A. Olofsson, "Epiphany-v: A 1024 processor 64-bit risc system-on-chip," *arXiv preprint arXiv:1610.01832*, 2016.

[17] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani, "A network on chip architecture and design methodology," in *Proceedings IEEE Computer Society Annual Symposium on VLSI. New Paradigms for VLSI Systems Design. ISVLSI 2002*. IEEE, 2002, pp. 117–124.

[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.

[19] J. Canny, "A computational approach to edge detection," *IEEE Transactions on pattern analysis and machine intelligence*, no. 6, pp. 679–698, 1986.

[20] M. Heath, S. Sarkar, T. Sanocki, and K. Bowyer, "Comparison of edge detectors: a methodology and initial study," *Computer vision and image understanding*, vol. 69, no. 1, pp. 38–54, 1998.

[21] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski, and J. Teich, "Electronic system-level synthesis methodologies," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1517–1530, 2009.

[22] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemCTM*. Springer Science & Business Media, 2007.

[23] "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, 2012.

[24] E. Arasteh and R. Dömer, "Fast loosely-timed deep neural network models with accurate memory contention," *ACM Transactions on Embedded Computing Systems (TECS)*, 2023.