UNIVERSITY OF CALIFORNIA,
IRVINE

A Loosely-Timed TLM-2.0 Model of a JPEG Encoder on a Checkerboard GPC

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Electrical Engineering

by

Arya Daroui

2022

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This thesis is built upon a foundation of knowledge and advice provided by Professor Rainer Dömer and his team—my colleagues. It is with humble thanks that I contribute to that foundation, hoping for it to be useful for those who come after me.

ABSTRACT OF THE THESIS

A Loosely-Timed TLM-2.0 Model of a JPEG Encoder on a Checkerboard GPC

By

Arya Daroui

Master of Science in Electrical Engineering

University of California, Irvine, 2022

Professor Rainer Dömer, Chair

Common, classical computer architectures are based upon few computational cores that collaborate and communicate through larger, slower system memory. In this work, we introduce a configurable, checkerboard grid of processing cells architecture with distributed cores and memories designed to maximize the benefits of parallelization. We explore the checkerboard model and a classical model at a high level to compare their behaviors in a moderately parallelized JPEG encoder application benchmark. The models are simulated with a Loosely-Timed, SystemC TLM-2.0 test platform with timing by processor core, memory, and memory controller, and transaction. Our experimental results show a 66% faster execution speed and higher memory bandwidth headroom for the checkerboard architecture, compared to the classical architecture.

Chapter 1

Introduction

Except for in exotic designs, classical computer architectures revolve around a processor that communicates with external devices and internal controllers to execute program data stored in memory, as shown in Figure 1.1. There have been modern developments expanding on and moving away from this base architecture, such as multiple processing cores [1], direct memory access [2], specialized processing units [3], and tiered memory [4], but designs generally still rely on the two main, discrete blocks of a computer system: the processor and its memory.

In this work, we evaluate a proposed *checkerboard **g**rid of **p**rocessing **c**ells* (GPC) [5] architecture with equal, distributed processing cores and memories that share resources locally,

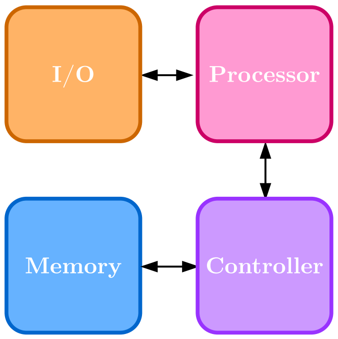

Figure 1.1: A high level view of classical computer architectures.

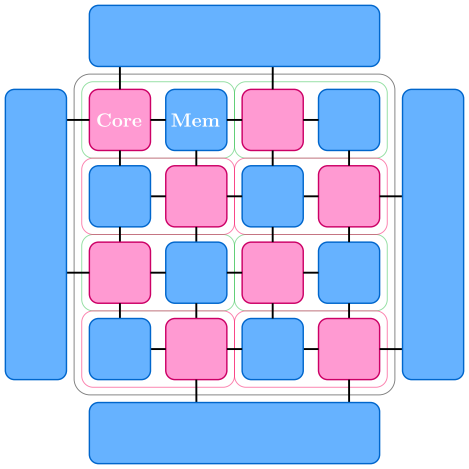

Figure 1.2: A simplified representation of the GPC [5].

illustrated in Figure 1.2. In contrast to classical designs, on the same chip, there are multiple core-memory pairs grouped into tileable cells; each cell can contain data unique to the core within the cell, and it can transfer the data to neighboring cells as needed. The goals of this distributed design are to take advantage of parallelizable programs, and to reduce the memory bandwidth issues found in classical architectures.

We will explore processor and memory utilization characteristics of the GPC and compare them to a ***c**lassical **m**ulti-core **a**rchitecture* (CMA) using simulated, system-level models. They will both be measured with a moderately parallelized JPEG encoder benchmark at varying processor and memory speeds in order to realize their bandwidth behaviors.

1.1 The Memory Bottleneck

In computer architecture, there lies a problem in that when the processor finishes a task and needs to access memory, it must wait as the data is transferred over the bus. The processor, being comparatively much faster than the data transfer time of the system memory, waits for many wasted cycles, effectively limiting processing throughput to that of memory throughput. Although it is not unique to the Von Neumann Architecture [6], this is known

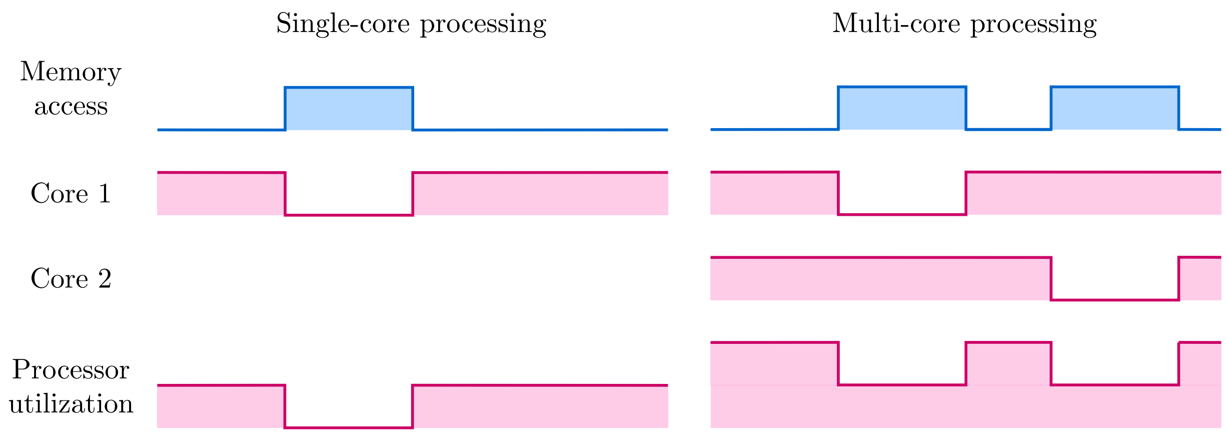

Figure 1.3: In a well synchronized program, the processor as a whole may not idle, but its individual cores do while waiting for memory access.

as the Von Neumann Memory Bottleneck [4]. There have been modern design advancements that mitigate the memory bottleneck, most notably: multi-core processing [7] and memory caching [4].

1.1.1 Multi-core processing

A typical computer program is written to be executed sequentially, where there is a single processing thread that executes instructions on data in order. As mentioned, when the processor needs to access to data in memory, it must wait for the transfer to complete before it can operate on the data. Likewise, while the processor is working, the memory is unused and waiting to be accessed. By adding another processing core, we can perform more memory accesses when the memory is idle and process parts of the program in parallel, increasing the memory and processor throughput together. This is illustrated in the simple timing diagram in Figure 1.3. Modern processors have begun to use this optimization heavily [7], and it is an inherent feature of our GPC.

While multi-core processing increases potential throughput for the processor, and well synchronized programs can avoid idling the processor as a whole, it still does not eliminate

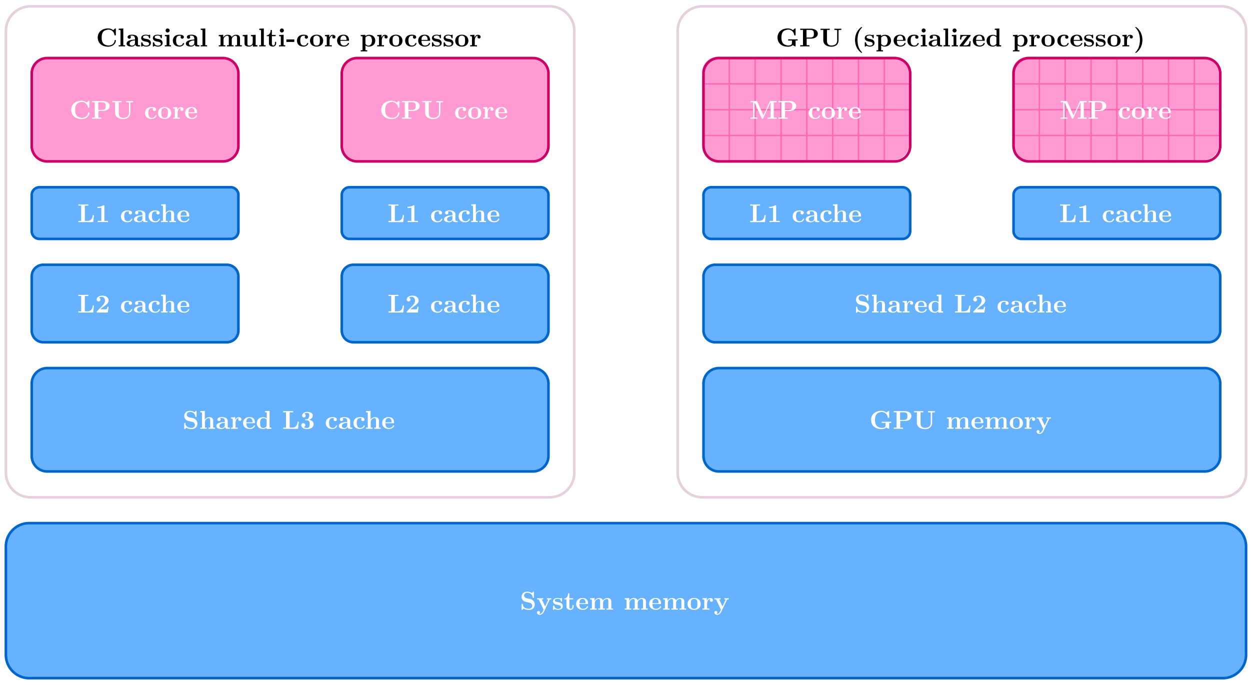

Figure 1.4: Tiered caches of memory are used for both general processors and specialized processors like GPUs.

idling for the component cores during memory access. It effectively pushes the memory latency issue down one level in the processor hierarchy. Further, multiple cores opens up contention issues in the cases that the memory accesses from the cores overlap.

1.1.2 Caching

Caching prefetches data it expects the processor will need and writes it onto smaller, faster, and more expensive tiered *caches* of memory near the core, shown in Figure 1.4. Caches' much faster access times reduce idle time for the processor; however, because of their small size, they must make smart guesses at what data the processor needs next. When there is an eventual cache miss, the memory access call goes up through each tier (level) of cache and up to system memory until it finds the data it needs.

We intentionally omit the usage of cache in our models of the GPC and the CMA. Caching is a fine-level implementation detail outside the scope of what we are exploring in this work, with prickly details such as instruction versus data cache that muddy the waters of our experiments. In terms of fairness, either architecture could employ cache[1], and there is no clear advantage provided to either architecture with or without it. With regard to model design, its effects are localized to memory access times; otherwise, it makes no difference to the program if the data is transferred from system memory or cache. So, while cache typically lies on the chip itself, we can abstract it away within the memory modules of our model, and keep our simulation at the system and transaction levels we desire.

1.1.3 System-level modeling

To simulate both our GPC and the CMA, we use SystemC [8], [9], a C++ library from Accellera Systems Initiative for system-level modeling [10]. SystemC, like popular hardware description languages, provides tools to easily specify and implement systems from the top down: starting with the specifications and constraints of how a system should operate, designing its functional modules in the hierarchy, and finalizing designs by connecting the communication channels between modules.

The significance of this work lies not just in the comparison between the GPC and CMA, but also in the development of the simulation platform. More specifically, our goal is to develop a generalizable, configurable, and accurate platform to simulate computer architectures using basic building blocks, and map real-world applications to them. It is from there we build our GPC and CMA architectures and run our experiments, verifying both the test platform and exploring the architectures' behaviors, in tandem.

We want to develop our platform at the system and transaction levels, meaning our SystemC modules will be composed of basic computer architectural blocks: external input-output,

[1]Each processing cell would have its own cache in the GPC.

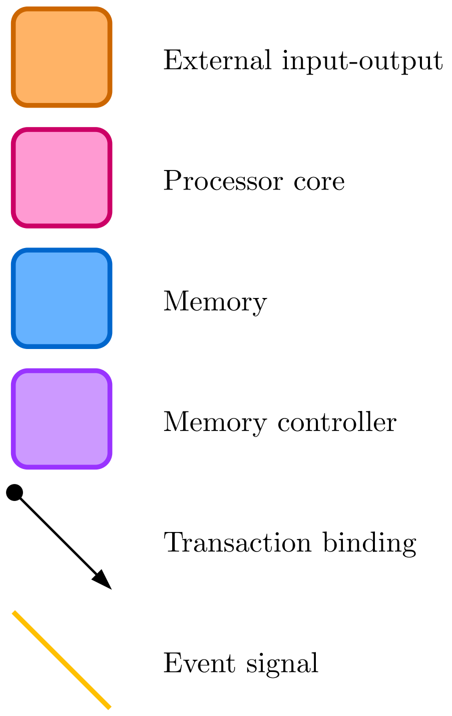

Figure 1.5: The basic building blocks of our system-level models, and their transaction.

processor cores, memories, and memory controllers[2]. The communication between these blocks is modeled with 'Loosely-Timed' transactions, as described by SystemC's TLM-2.0 standard [9]. This is summarized and color coded in Figure 1.5, which holds for all architectural figures presented in this work.

[2]Multiplexers and demultiplexers.

Chapter 2

JPEG encoding

The JPEG encoding is a common media file format for image compression [11]. Its popularity, utility, and moderate parallelizability make it a good choice to use as a benchmark for our simulations. The JPEG standard covers a variety of implementations, but we will use the typical implementation shown in Figure 2.1.

2.1 Color space transformation

We start with a simple bitmap color image as input, composed of red, green, and blue (RGB) channels for pixel intensity, and read 8×8 blocks of pixels. We send each block into a color space transformation function that converts it from RGB to Y, Cb, and Cr (YCC) channels,

$$\begin{bmatrix} Y \\ C_\mathrm{b} \\ C_\mathrm{r} \end{bmatrix} = \begin{bmatrix} 0 \\ 32768 \\ 32768 \end{bmatrix} + \frac{1}{256} \begin{bmatrix} 77 & 150 & 29 \\ -44 & -87 & 131 \\ 131 & -110 & -21 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \tag{2.1}$$

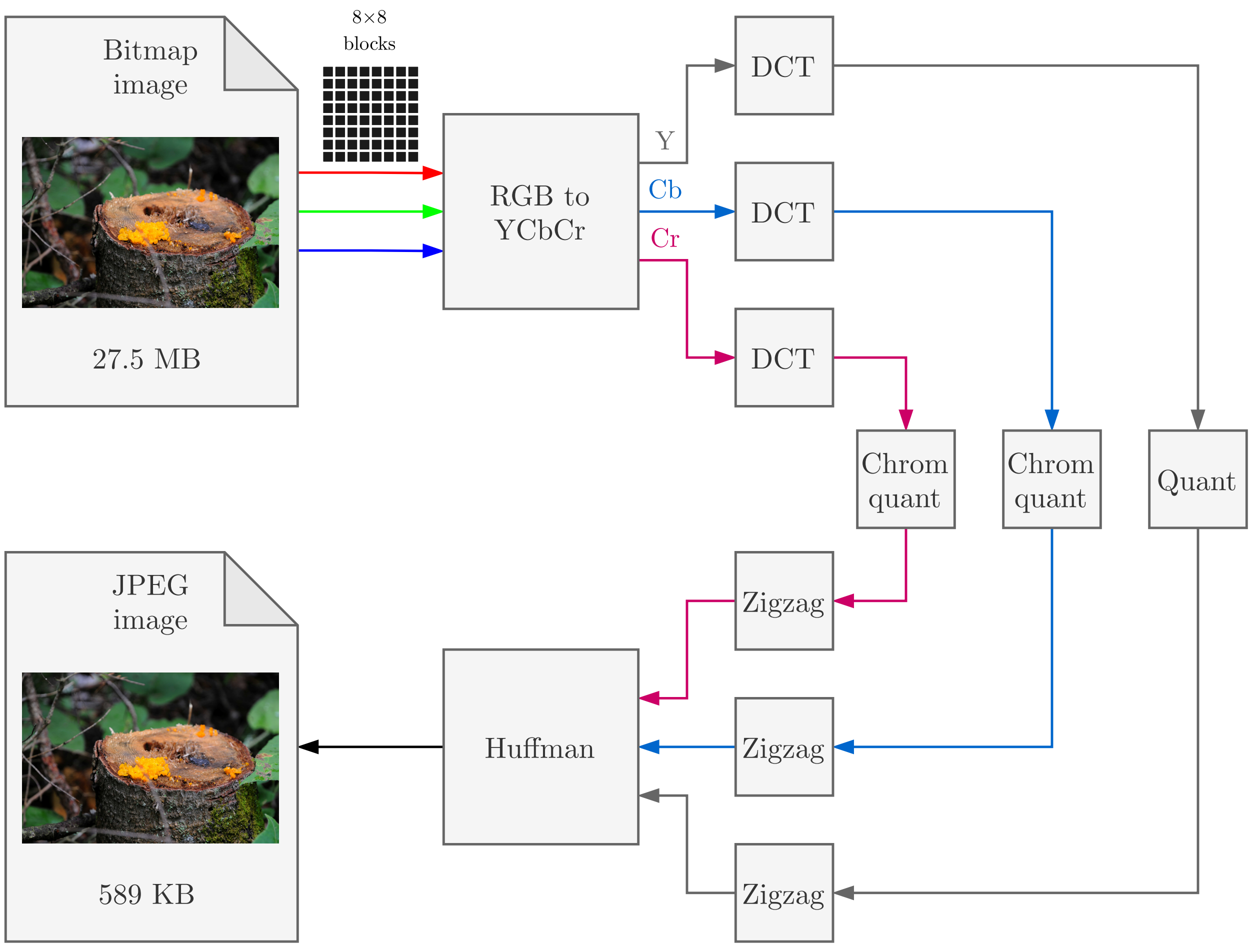

Figure 2.1: The JPEG encoder and its functional stages, parallelized by color channel.

The Y channel describes grayscale luminosity which human eyes are more sensitive to, and the Cb and Cr describe blue and red chromaticity, relative to green. We continue to operate by 8×8 blocks throughout the encoding.

2.2 DCT

For each block, we apply the discrete cosine transform (DCT), which yields a block of coefficients corresponding to the frequency components of the block, with the lowest frequency at the top left, and the highest at the bottom right, shown in Figure 2.2. The human eye is more sensitive to low frequencies, so the next step is to selectively drop high frequency components, which is done through quantization. The DCT stage is mildly lossy due to rounding in the calculation.

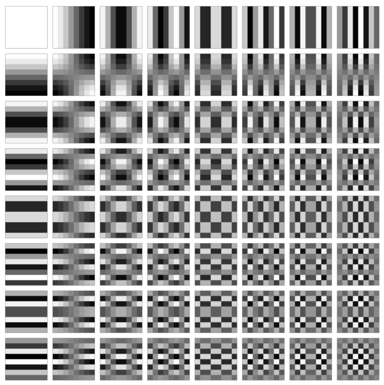

Figure 2.2: DCT component representation [12]. The lowest frequencies are represented at the top left, and the highest frequencies at the bottom right.

2.3 Quantization

The quantization process works by dividing (rescaling) the value of each DCT component by a corresponding value in a given quantization table and rounding the quotient. This rounding causes most of the perceivable quality loss JPEG is known for[1]. There are two different quantization tables used, one for the luma channel, and one for the chroma channels. These are given in Table 2.1 and are defined by the JPEG standard for a quality level of 50 [11].

Table 2.1: 8×8 quantization divisors for luma and chroma channels.

Luma	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Chroma**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	16	11	10	16	24	40	1	61	**1**	17	18	24	47	99	99	99	99
2	12	12	14	19	26	58	0	55	**2**	18	21	26	66	99	99	99	99
3	14	13	16	24	40	57	9	56	**3**	24	26	56	99	99	99	99	99
4	14	17	22	29	51	87	0	62	**4**	47	66	99	99	99	99	99	99
5	18	22	37	56	68	109	103	77	**5**	99	99	99	99	99	99	99	99
6	24	35	55	64	81	104	113	92	**6**	99	99	99	99	99	99	99	99
7	49	64	78	87	103	121	120	10	**7**	99	99	99	99	99	99	99	99
8	72	92	95	98	112	100	103	99	**8**	99	99	99	99	99	99	99	99

2.4 Zigzag

Because of the previous quantization step, the highest frequency components in the bottom right of the block are are typically zeroed out. The encoder takes advantage of this with run length encoding, where the encoder stores how many ones or zeros there are in a consecutive sequence, rather that storing the list of ones and zeros itself. However, to do this, the encoder must index diagonally instead of by row or column, which it does by the zigzag pattern illustrated in Figure 2.3. The compression from the zigzag run length encoding is lossless.

[1]Chroma subsampling is another considerable factor of perceivable quality loss, but it is not implemented in our encoder.

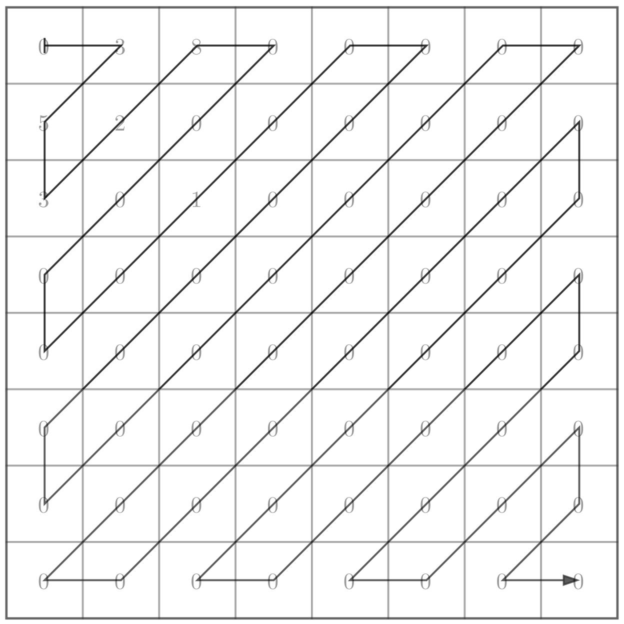

Figure 2.3: The zigzag pattern used to diagonally index and encode the block.

2.5 Huffman entropy encoding

The previous three steps were separable by channel, and could be processed in parallel. But now each channel is combined and compressed with Huffman encoding [13], [14]. Huffman encoding works by representing variable lengths of data with known prefixes that are stored in their place. These prefixes are determined by the frequency of symbols in the data, and are predefined by the JPEG standard; their tables have been omitted due to their size.

Chapter 3

Checkerboard architecture

In contrast to a classical multi-core processor that would be connected through the system bus to system memory, the checkerboard GPC is characterized by its distributed, alternating core-memory pairs. In this chapter, we will describe the features of this design, the addressing scheme and communication protocols needed to make it work, the layout and timing of the model, and the user interface for mapping applications.

3.1 Design features

There are three main features to the GPC that distinguishes it from the classical architecture: distributed memory access, distributed processing, and scaleability.

3.1.1 Distributed memory access

The greatest difference to classical designs is the checkerboard's distributed memory, which negates the need for a central system bus. Not unlike low-level cache, each core has a dedicated

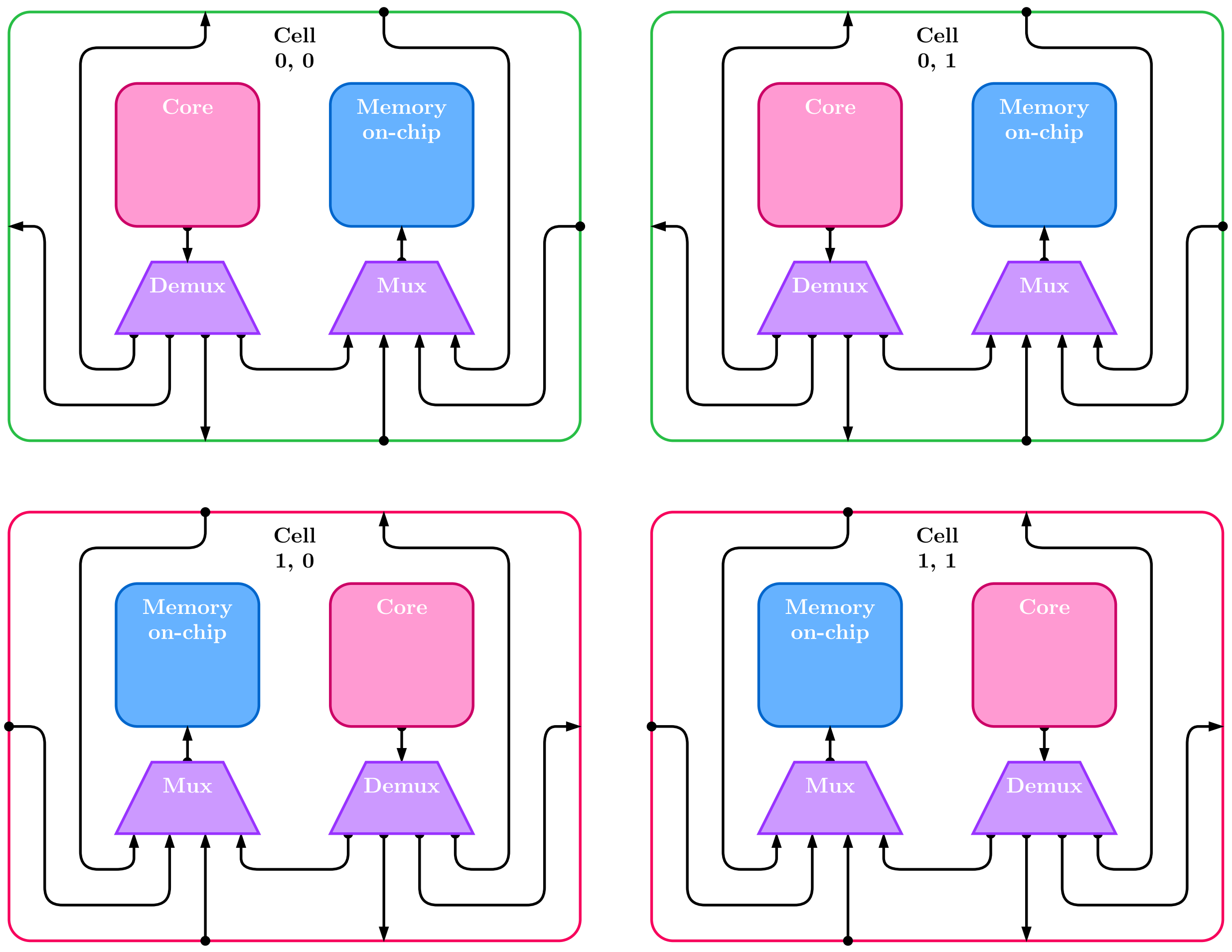

Figure 3.1: Component cells of the checkerboard. Red cells are 'right-handed', and green cells are 'left-handed' to make a tileable pattern.

memory unit beside it in its processing cell, and has direct access to slightly farther away memories in neighboring cells, illustrated in Figure 3.1. The combination of proximity and unimpeded access is expected to yield significantly lower memory latency and less contention.

3.1.2 Distributed, parallel processing

The checkerboard architecture is inherently multi-core, and therefore, conducive to parallelization. As mentioned previously, parallel processing can yield significant performance benefits to program execution by distributing workload over cores, and reducing processor idle time.

An additional, unique aspect of the checkerboard is that because each core only communicates with memory units neighboring it, the program function and memory access can be further encapsulated, making the development of pipelined, parallelized programs more intuitive. For example, for our JPEG encoder, each functional stage of the encoder is isolated to a single core, and reads in only the data it needs, and writes out the data it has processed to its neighboring memory.

3.1.3 Scaleability

The checkerboard, being composed of smaller computational cells can be scaled with more or fewer cores to meet user specification without changing the overall design. The scalability is intuitive to see in Figure 1.2 and Figure 3.1, but to be put explicitly, there are only two patterns to be mindful of:

1. Cell orientation alternates by row.
2. A memory on the edge of the checkerboard will have an open port for each edge it borders.

Cell orientation refers to whether the core will be on the left or right of the cell. This alternation allows more direct access between cell rows without crossing over connections.

3.2 Model layout

Our model for the the checkerboard is broken into several hierarchical layers using SystemC modules. Modules serve as a way to encapsulate the core functionality for a part of a system so that it can be easily reused, re-encapsulated, and abstracted. They are analogous to the generic functional blocks in block diagrams. The layout of the GPC with its component modules are shown in Figure 3.2.

The GPC model has three main hierarchical layers: the top level with the checkerboard chip, off-chip memory, controllers, and external input-output; the checkerboard level with its many processing cells; and processing cell level with the core-memory pairs and their memory controllers. This hierarchy is shown in Figure 3.3.

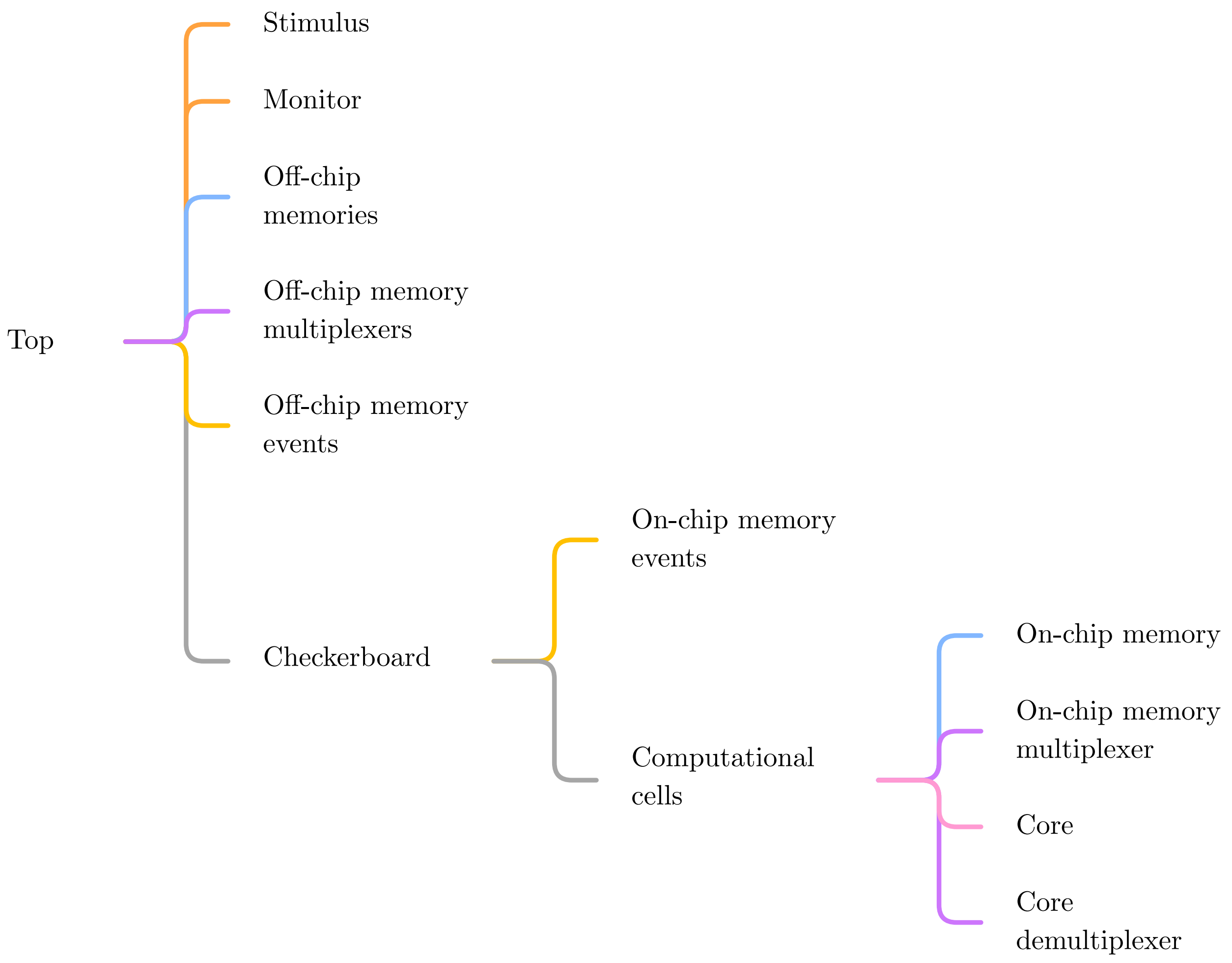

Figure 3.3: The hierarchy of the checkerboard GPC's component modules.

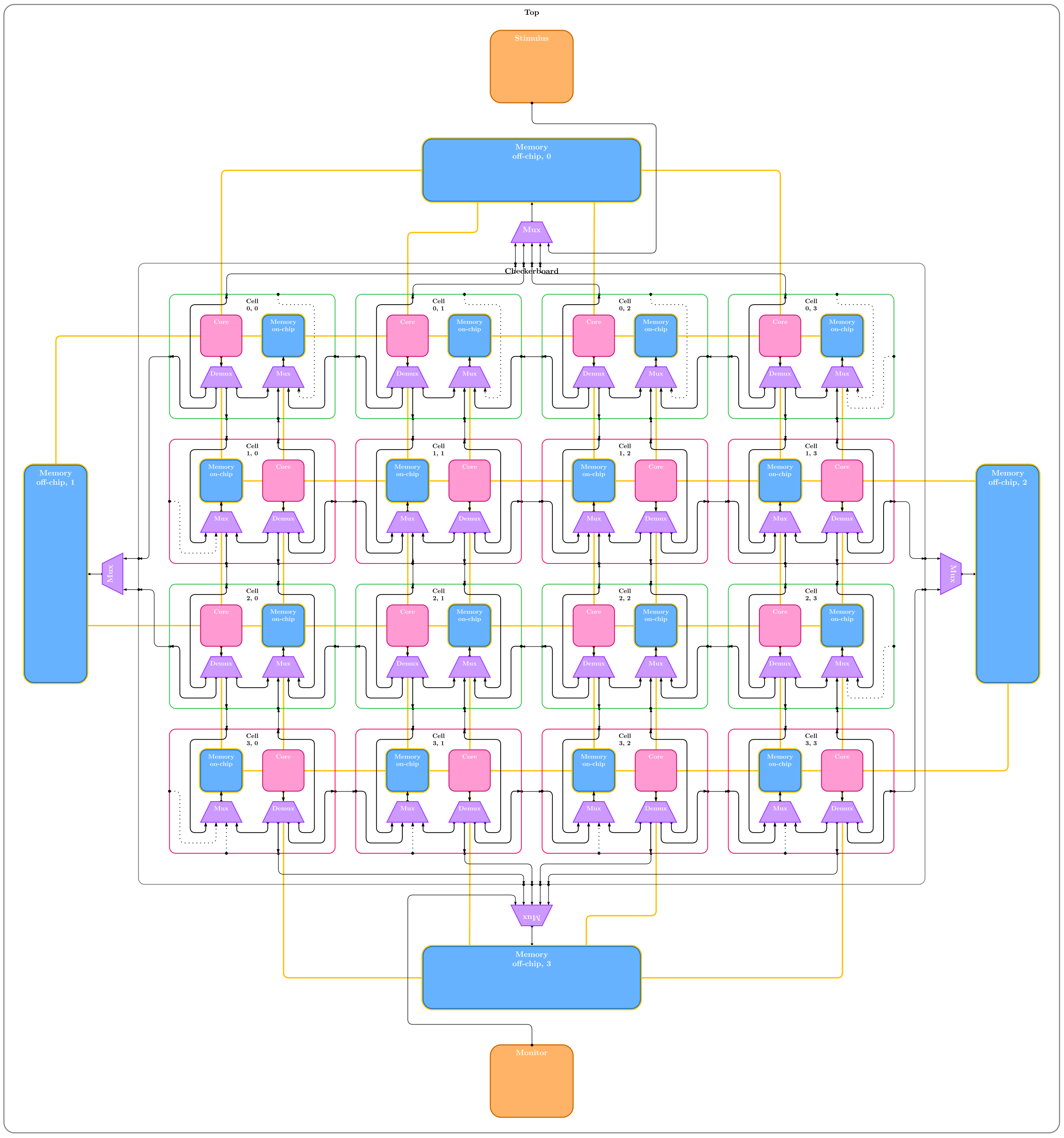

Figure 3.2: A detailed view of the checkerboard model.

3.3 Communication

Modules communicate with each other using TLM-2.0 socket objects which are bound between modules; each socket is either an initiator or target for communication transactions. For our GPC, the transactions are composed of a read or write command, initiating address, target address, data length, and success status; there is delay data passed alongside the transaction for simulation timing. Additionally, each memory module has an associated *event* signal, which is used for communication synchronization on both read and write. For example, when trying to share data between cores, a core module will send a write transaction to a memory module. After the write is complete, the core module will use an event to signal a waiting, receiving core that it can send a read transaction for the data that was just written.

3.4 Address space

Even though the GPC heavily utilizes local memory access, there is a defined, absolute address space; it does not use a relative addressing scheme between cells. The currently implemented model specifies a 32-bit machine with a maximum of four rows by four columns of cells, and four off-chip memories. The 4 GB memory capacity is halved between the on-chip and off-chip memories, where the maximum size of each memory is 128 MB for on-chip, and 512 MB for off-chip. The address map is shown in Figure 3.4.

The most significant bit determines if the address is on-chip or off chip. For on-chip memory,

Bit	31	30	29	28	27	26	25	24	23	⋯	0
On-chip memory	0	row		col		rest of address					
Off-chip memory	1	pos		rest of address							

Figure 3.4: Address map of the checkerboard.

the ‘row’ and ‘col’ are the bits that determine the row and column of the cell. For off-chip memory, ‘pos’ are the bits determine the position, with 00 as upper, 01 as left, 10 as right, and 11 as lower.

3.5 Timing

In our experiments, our focus is to analyze the interaction between processor delay and memory latency, for which the main measure is simulated execution time. To do this, our SystemC model has timing annotations that simulate and combine delay,

1. per core (processor speed),
2. by memory type,
3. by transaction size,
4. by memory controller (multiplexer).

Each core’s delay depends upon the functional block programmed onto the core, and memory latency will be different depending on whether the unit is on or off of the chip, and whether it is being read from or written to. To delay by transaction size, we consider the size of the data bus (typically the same as the word size) and the burst length of the memory. Our level of abstraction sits above the details of memory clock speed and CAS latency, so we generalize this by adding up memory latency per chunk of word size × burst length that fits in the transaction,

$$\text{transaction delay} = \left(\text{incfloor}\left(\frac{\text{transaction size}}{\text{word size} \times \text{burst length}}\right) + 1\right) \times \text{memory latency}. \quad (3.1)$$

The incfloor(x) function is the same as a normal floor(x) function except the right edge of each step is inclusive[1], preventing an extra access delay if the data size is a perfect multiple of the chunk. For example, if we have a transaction that is 1024 bits wide and a burst length of four,

$$\begin{aligned} \text{transaction delay} &= \left(\text{incfloor}\left(\frac{1024}{32 \times 4}\right) + 1\right) \times \text{memory latency} \quad (3.2) \\ &= 8 \times \text{memory latency}. \quad (3.3) \end{aligned}$$

Without considering transaction size, it would take one memory latency delay to execute the entire 1024 bit transaction; without considering burst length, it would take 32 memory latency delays; both are less realistic.

In addition to these individual delays, we also employ memory multiplexer timing delay and contention from Malekzadeh and Dömer [15]. Contention is critical in accurately simulating the dynamic effects different timing delays have on resource utilization and simulated execution time. It ensures only one transaction and its associated delays are occupying the memory controller at a time. If multiple transactions are received, they are performed in FIFO order, and the waiting transaction will induce additional idling for the cores. As can be seen in Figure 3.5, without contention, even though the delays of individual transactions are processed, the memory would be performing multiple memory accesses concurrently, potentially even multiple from the same core, which is unrealistic both in terms of execution time, and data volatility.

[1] I.e., it will round a perfect integer down, incfloor(8) = 7.

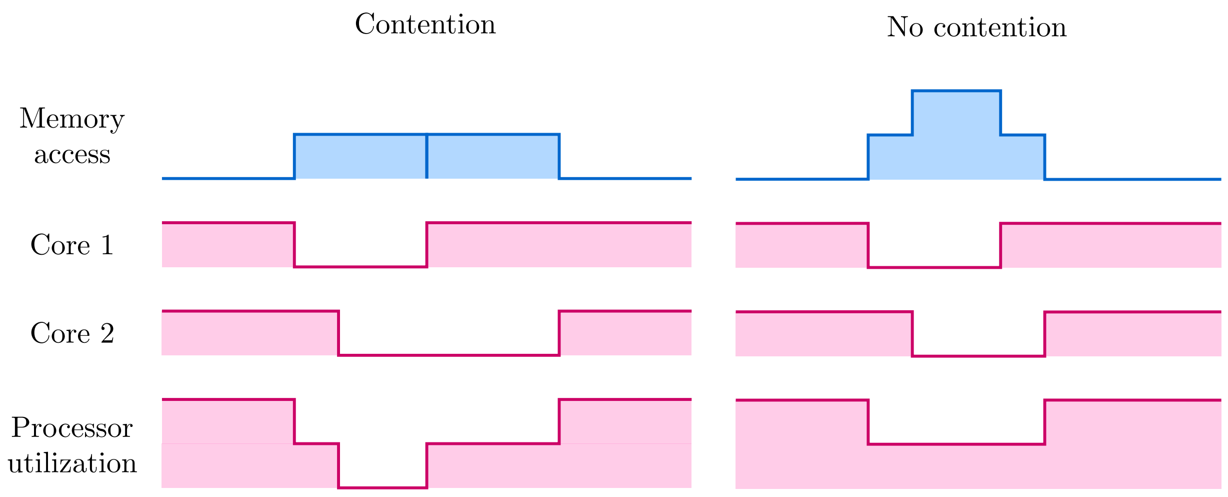

Figure 3.5: With contention on, overlapping memory access transactions must wait for each other to finish. Without contention, they can all simultaneously occupy the memory controller.

3.6 User interface

In our goal of making this generalizable platform, it is important to have an intuitive interface for when the user constructs and tests their own models and applications. For the checkerboard, the user only needs to interact with one file that dictates the processing loop for each core. When mapping a generic program, the user would divide the program into independent, component parts, and place them in the provided `main()` functions for each core, with care taken for the core's position and its memory connections, like in Code block 3.1. The only additional code would be to write to and read from the adjacent memory addresses, which is simplified to single function calls with the provided MemTools memory sharing library. Whereas one would have to manually record each location in the address space for their data and manage the event signals, the MemTools library keeps track of addresses and signals events for the automatically. This is described in greater detail in chapter 4.

Code block 3.1: How the user programs a core.

```cpp
// ... For each core used,
class Core12: public Core {
  public:
  void main(void) {
    // Main program loop goes here
    // Should read input from neighoring memory at start
    // And write to output memory at end
  }

  Core12(sc_module_name n,
        int _y,
        int _x,
        sc_event & _S_UP,
        sc_event & _S_LEFT,
        sc_event & _S_RIGHT,
        sc_event & _S_DOWN):
    Core(n, _y, _x, _S_UP, _S_LEFT, _S_RIGHT, _S_DOWN) {
    SC_THREAD(main);
  }
};

// ... Each core is bound to its cell, the DUT, and the top
   module. This is given to the user
```

Chapter 4

Memory sharing protocol

Because of the direct data sharing between cores, the memory modules act more as channels for communication, rather than generic data storage. Consequently, it is helpful to use a standard data transfer and synchronization protocol to prevent common multi-threaded pitfalls like data overwrites and deadlock. The protocol used is a simple FIFO queue with synchronized counters, and its tooling is provided in the given MemTools library, allowing the user to simply push and pop from memory, without having to worry about individually tracking addresses and events. The MemTools user interface is shown in Code block 4.1, with example code in Code block 4.2.

4.1 The memory map and partitions

The memory map used by MemTools is illustrated in Figure 4.1. The shared memory is split into two portions, the counters and the queue. Counters keep a copy of how many slot positions have been sent and received in the queue. When pushing data to a memory module, the counters are checked to see if the queue is saturated. If so, it waits for a signal from the

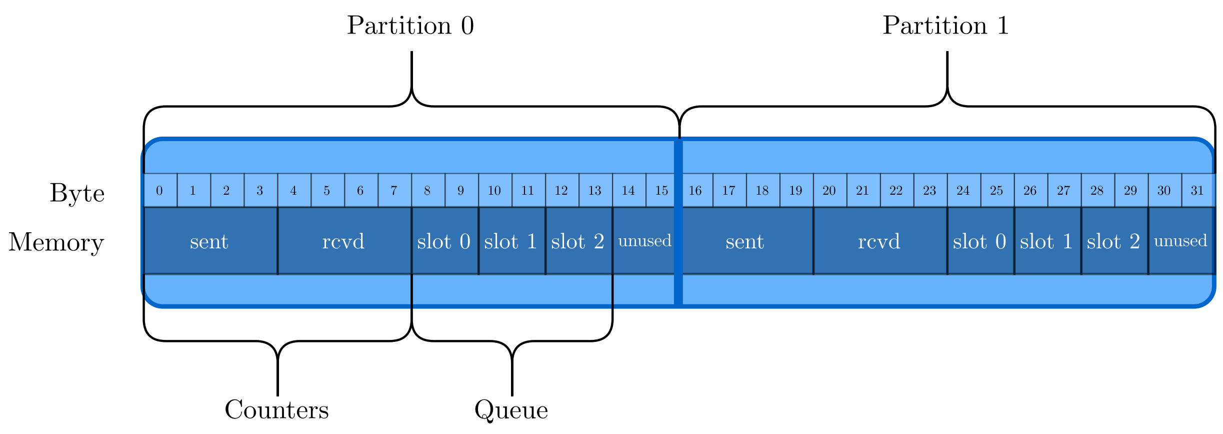

Figure 4.1: The memory map used for the FIFO protocol.

receiving queue that it has popped the slot. Once a slot is available, it is overwritten, the receiving core is signaled, and the counters updated.

It can sometimes be useful to use a single memory for multiple channels and cores. When this is the case, the memory is divided into equal partitions, with each partition adhering to the memory map outlined above. The most typical multi-channel patterns are to separate, combine, and parallelize data flow, as shown in Figure 4.2. The lattermost is done in the first stage of the JPEG encoder for the RGB channels. These figures show only two partitions, but the number of possible partitions in the protocol is limited only by the memory space needed to manage the counters and the queue. Additionally, the number of slots in the queue is adjustable, leaving unused memory in the partition, if desired.

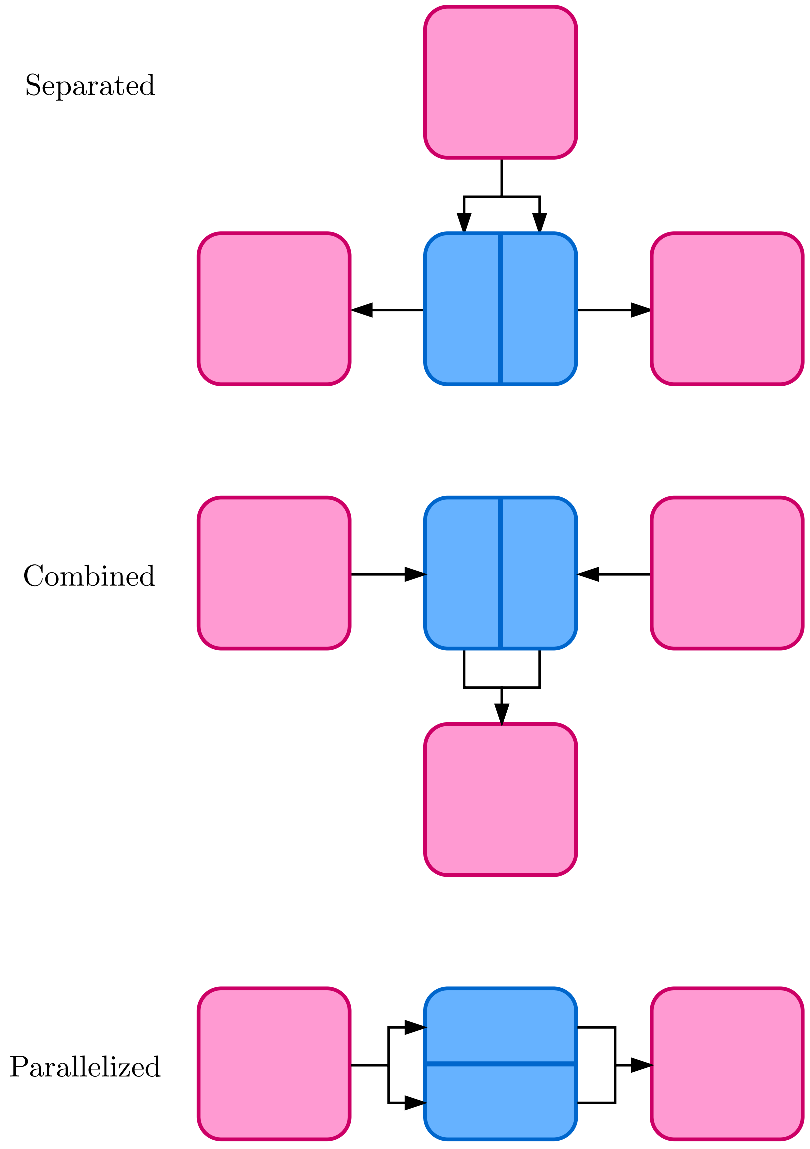

Figure 4.2: The memory map allows for data splitting from one processing core to many, from many processing cores to one, and for parallel passthrough.

Code block 4.1: The interface for MemTools.

```
// MemTools public (user-facing) method signatures
MemTools(tlm::tlm_initiator_socket<> &socket,  // Constructor
         sc_event &signal,
         unsigned startAddr_,
         unsigned memSize_,
         unsigned numPartitions_,
         unsigned slotLen_,
         unsigned numSlots_);
void PushData(void *data,
              unsigned dataLength,
              unsigned partitionIndex);
void PopData(void *data,
             unsigned dataLength,
             unsigned partitionIndex);
```

Code block 4.2: How the user programs memory access.

```
// In the user's checkerboard code:
class Core12: public Core {
  public:
  void main(void) {
    int in_data;
    int out_data;

    MemTools memIn(CoreBus, sig_up, id_to_memAddress(0, 2),
       MEMORY_SIZE, 1, sizeof(int),  NUM_SLOTS);
    MemTools memOut(CoreBus, sig_down, id_to_memAddress(2, 2),
       MEMORY_SIZE, 1, sizeof(int),  NUM_SLOTS);

    while(1) {
    // Read from memory
    memIn.PopData(&in_data, sizeof(int), 0);

    // Data processing goes here

    // Write to memory
    memOut.PushData(&out_data, sizeof(int), 0);
    }
  }
  // ...
};
```

Chapter 5

Application mapping

We have mapped the JPEG encoder application to both the checkerboard, and a classical multi-core architecture with parallelization by channel, and pipelined by the functional blocks in Figure 2.1. They are composed of the same building blocks described in Figure 1.5 and both use the MemTools memory sharing protocol; the only difference is their architectural layout.

5.1 Checkerboard

For the checkerboard, as seen in Figure 5.1, we chose the 4×4 implementation per the current maximum specification. This lead to an interesting side effect in our mapping, where we had to utilize an extra cell that does nothing but forward data to the next core. Because our YCC block pushes into three channels, and the Huffman block pops from three memories, there is no arrangement such that the 3 blocks in three channels all meet, surrounding the targeted Huffman block. The forwarding block will incur extra memory delay, but no extra processing delay. The stimulus and monitor blocks are considered external to our experiments and have

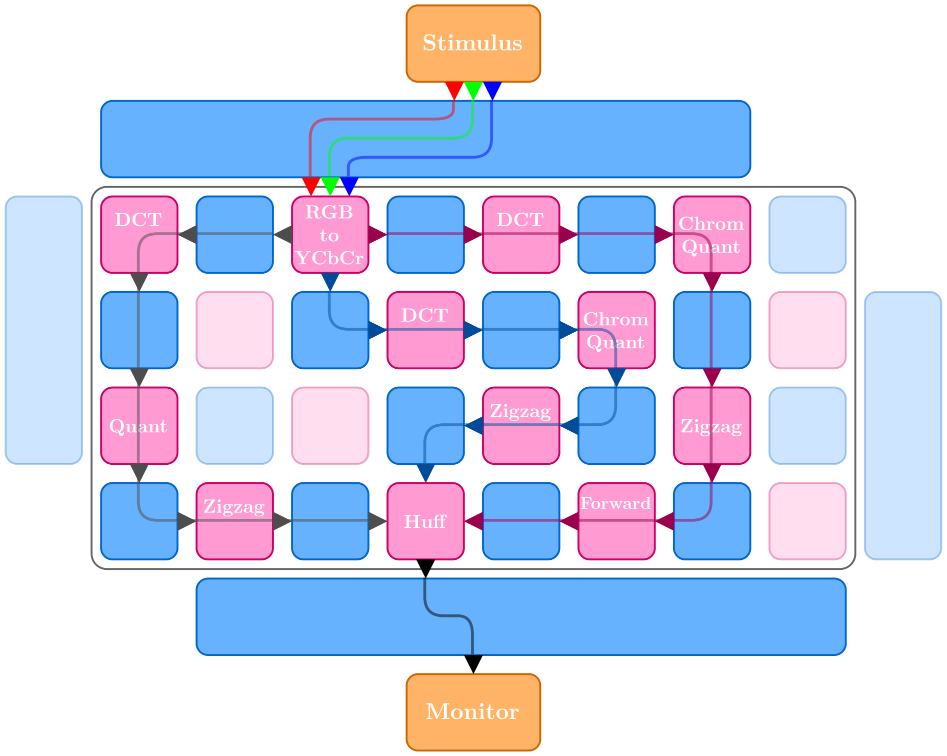

Figure 5.1: The mapping of the JPEG encoder application to the checkerboard GPC.

no timing annotations to incur any delay.

5.2 Classical multi-core

On the classical multi-core architecture, we had 11 cores for the 11 functional blocks, all connected to a single multiplexer that interfaces with a single system memory with an event system connected to every block, as shown in Figure 5.2 and mapped in Figure 5.3. Similar to the GPC, the stimulus and monitor do not incur any processing delay that is counted in the simulator. However, it also does not need or use the extra forwarding block.

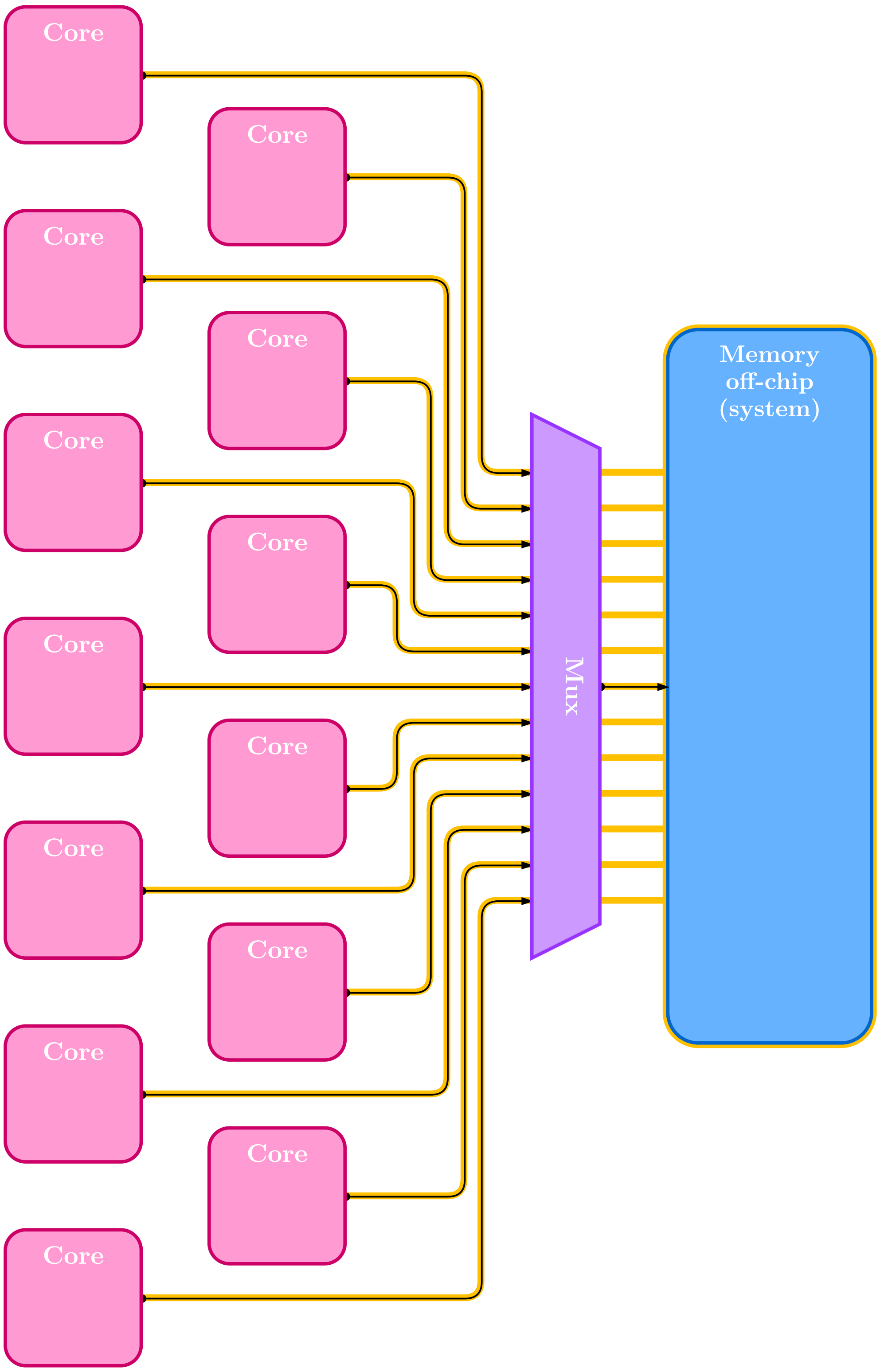

Figure 5.2: The CMA architecture.

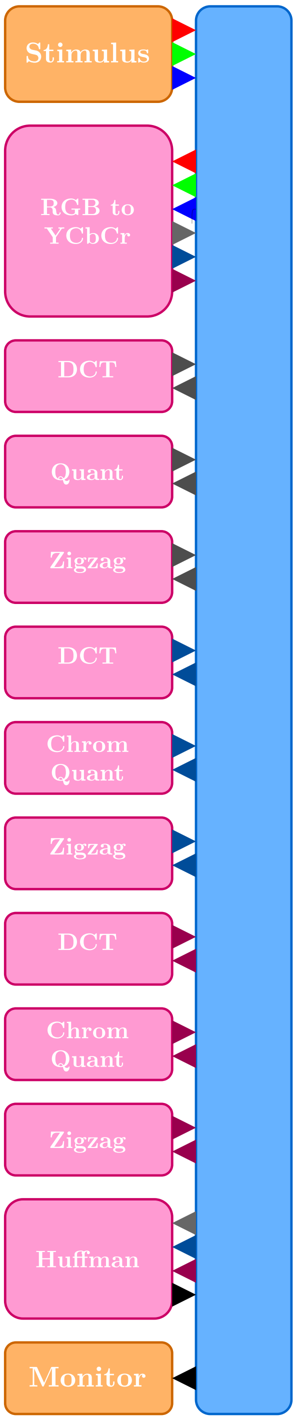

Figure 5.3: The mapping of the JPEG encoder application to the CMA.

Chapter 6

Experiments and results

The goals of our experiments are to find how sensitive the CMA and GPC architectures are to memory and processor bottlenecks, and to verify that our test platform yields reasonable results. Our main experimental metric is the simulated execution time for encoding a JPEG image, as detailed in chapter 2, while changing the clock speed and memory latency of the model. We will dive deeper into the setup of these experiments and their accompanying results.

6.1 Setup

As mentioned, there are only two parameters we adjust: clock speed and memory latency. To make sure we evaluate a wide breadth of possible scenarios, we started with a standard, practical value for each, and evaluate at 0.25×, 0.5×, 1×, 2×, 4× that value. Lastly, we evaluate at the extreme cases of untimed processing delay and no memory latency.

Table 6.1: Processing delay times for the JPEG application at 1.0 GHz.

Block	**Delay (μs)**
RGB to YCbCr	2.336
DCT	2.164
Quantize	1.431
Zigzag	1.013
Huffman	1.014

6.1.1 Clock speed (processing delay)

The clock speed of the processor is emulated by appropriately choosing the timing delay within the program of each core. For our JPEG encoder with each of its stages mapped to a core, we found the timing for the single 8×8 pixel block each stage operates on. The timings were recorded from a real-world 3.0 GHz x86 processor [16], which we scaled down to 1.0 GHz for our standard value so that we have more range when varying the parameter. The timings are shown in Table 6.1.

6.1.2 Memory latency

The memory latencies chosen for the simulation were based off of estimations of real-world values for each type of hardware. The CMA's system memory and the GPC's off-chip memory are both expected to be using DRAM technology and we assume a latency of 70 ns [17, p. 378]. For the GPC's on-chip memory, we assume SRAM-like performance and give it a 4 ns latency [17, p. 378]. Because the GPC predominantly uses on-chip memory to communicate between cores, this immediately puts it at an advantage. For a more fair comparison between the architectures, we also considered a memory latency of 70 ns for the on-chip memory, which we denote as GPC Memory-matched (GPCMM). Lastly, we assumed a multiplexer latency of 4 ns [18], kept contention on, and set a burst size of four. We do not alter the multiplexer's latency when varying memory latency. These values are tabulated in Table 6.2.

Table 6.2: Memory access latency induced by each device in nanoseconds.

	CMA	**GPC**	**GPCMM**
Off-chip memory	70.0	70.0	70.0
On-chip memory	-	2.5	70.0
Multiplexer	4.0	4.0	4.0

6.1.3 Extreme parameters

We also run our experiments at the extreme values of no processing delay, and no memory latency. Each represent an important behavior that we build off of in our discussion of the results. This is illustrated by the simple timing diagram in Figure 6.1. More specifically, when the memory latency is set to zero, it represents the maximum performance capable of the processor because the memory instantly reads and writes the necessary data. In other words, it is 100% processor bottlenecked, 0% memory bottlenecked, and represents the theoretical ideal of the *application* in terms of minimum execution time for a given clock speed. We expect this to yield the same results across all architectures since they are all running the same application.

When there is no processing delay (we refer to this more briefly as 'untimed'), the processor effectively has infinite clock speed. It instantly computes the input data and outputs it to the next stage. This represents the maximum possible *memory throughput* of the the architecture; it is 100% memory bottlenecked, and practically removes the application as a parameter of execution time. This will provide a measure of how the configurations compare at total memory access saturation.

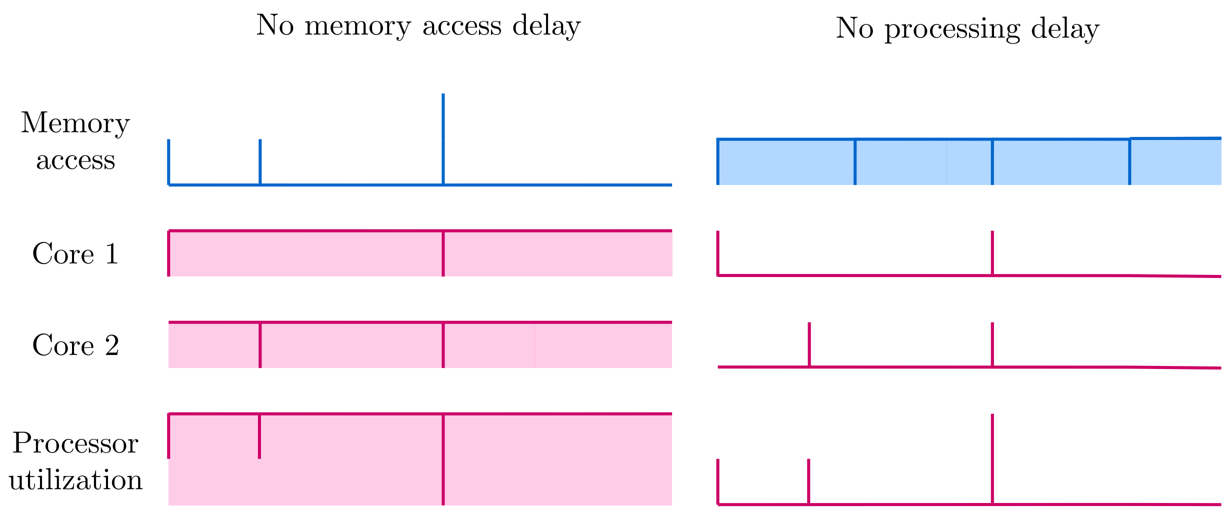

Figure 6.1: Timing diagrams for no processing delay, and no memory access delay.

6.2 Results

Our experiments on three architecture configurations yielded three tables of data, shown in Table 6.3, Table 6.5, and Table 6.4 that we will draw all of our plots from. First, we wish to draw direct comparisons between the CMA and the GPCMM.

The results of Table 6.5, and Table 6.4 have been projected onto a 3D surface in Figure 6.2. Cross sectional plots of this surface are plotted in Figure 6.3; the top row shows the the simulated times for the CMA, the bottom row shows the simulated times for the GPCMM, and the middle row shows their middling traces superimposed. From the 3D surface, we find that the GPCMM is faster than the CMA under most similar conditions, and especially as the processor and memory slow down. From the cross-sectional plots, we find that the GPCMM enjoys moderate gains as the clock speed increases, but has some diminishing returns, although not as severe as the CMA. Additionally, as the memory latency slows (increases), the GPCMM takes a slight hit to its simulated execution times compared to the drastic increase for the CMA.

Table 6.3: Simulated execution times for the CMA in seconds.

		Memory latency					
		0 ns	**17.5 ns**	**35 ns**	**70 ns**	**140 ns**	**280 ns**
Clock speed	**250 MHz**	1.31	1.32	1.34	1.40	1.59	2.21
	500 MHz	0.65	0.67	0.70	0.79	1.10	1.92
	1 GHz	0.33	0.35	0.40	0.55	0.96	1.87
	2 GHz	0.16	0.20	0.28	0.48	0.93	1.84
	4 GHz	0.08	0.14	0.24	0.47	0.92	1.82
	Untimed	0.00	0.11	0.23	0.45	0.90	1.80

Table 6.4: Simulated execution times for the GPCMM in seconds.

		Memory latency					
		0 ns	**17.5 ns**	**35 ns**	**70 ns**	**140 ns**	**280 ns**
Clock speed	**250 MHz**	1.31	1.32	1.33	1.35	1.40	1.49
	500 MHz	0.65	0.67	0.68	0.70	0.75	0.90
	1 GHz	0.33	0.34	0.35	0.37	0.45	0.68
	2 GHz	0.16	0.18	0.19	0.23	0.34	0.56
	4 GHz	0.08	0.09	0.11	0.17	0.28	0.51
	Untimed	0.00	0.03	0.06	0.11	0.23	0.45

Table 6.5: Simulated execution times for the GPC in seconds.

		Memory latency					
		0 ns	**17.5 ns**	**35 ns**	**70 ns**	**140 ns**	**280 ns**
Clock speed	**250 MHz**	1.31	1.31	1.31	1.31	1.31	1.33
	500 MHz	0.65	0.65	0.66	0.66	0.66	0.74
	1 GHz	0.33	0.33	0.33	0.33	0.37	0.51
	2 GHz	0.16	0.16	0.16	0.19	0.25	0.41
	4 GHz	0.08	0.08	0.09	0.13	0.21	0.38
	Untimed	0.00	0.02	0.05	0.09	0.18	0.36

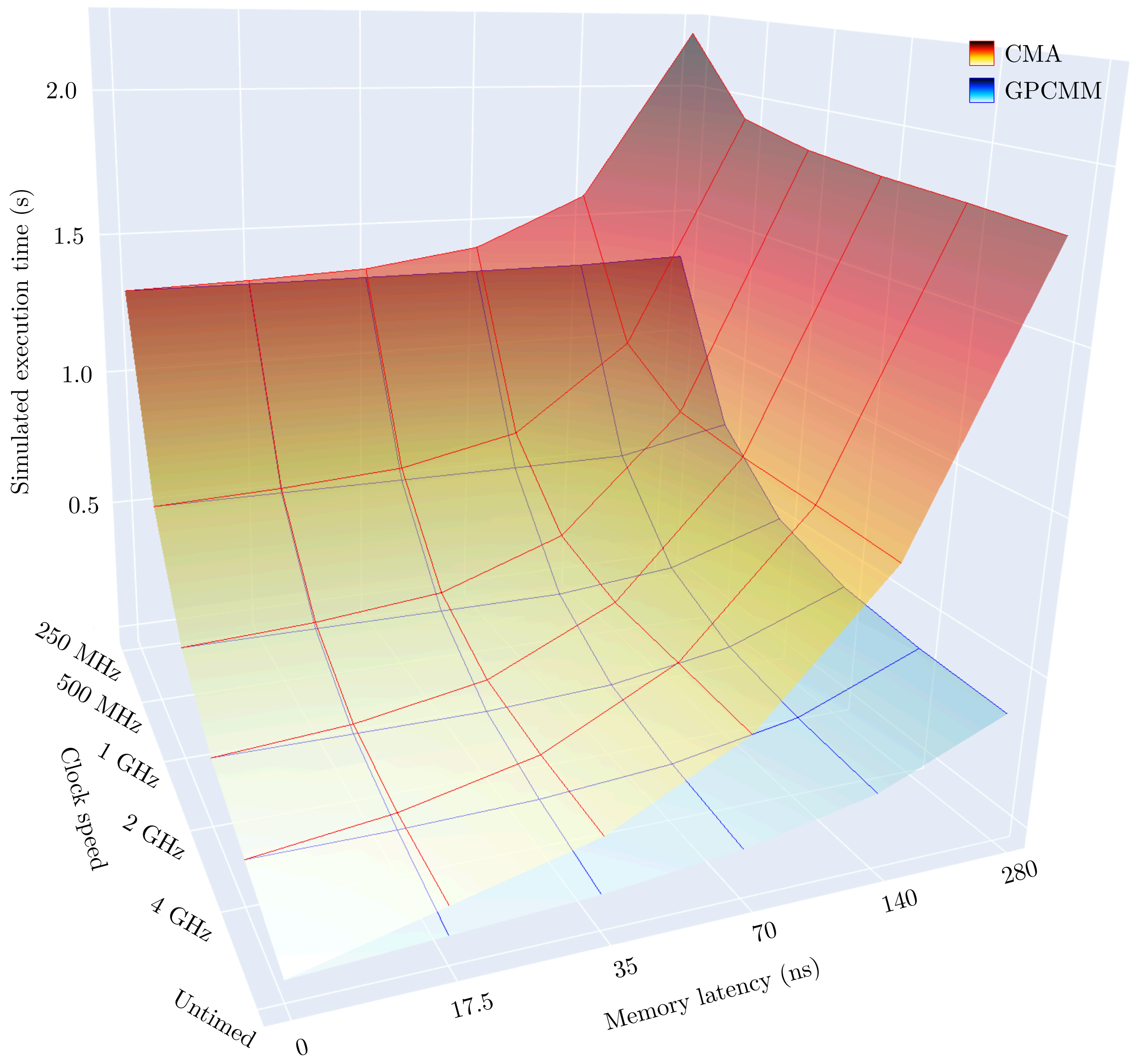

Figure 6.2: Simulated execution times of the CMA and GPCMM by processor speed and by memory latency.

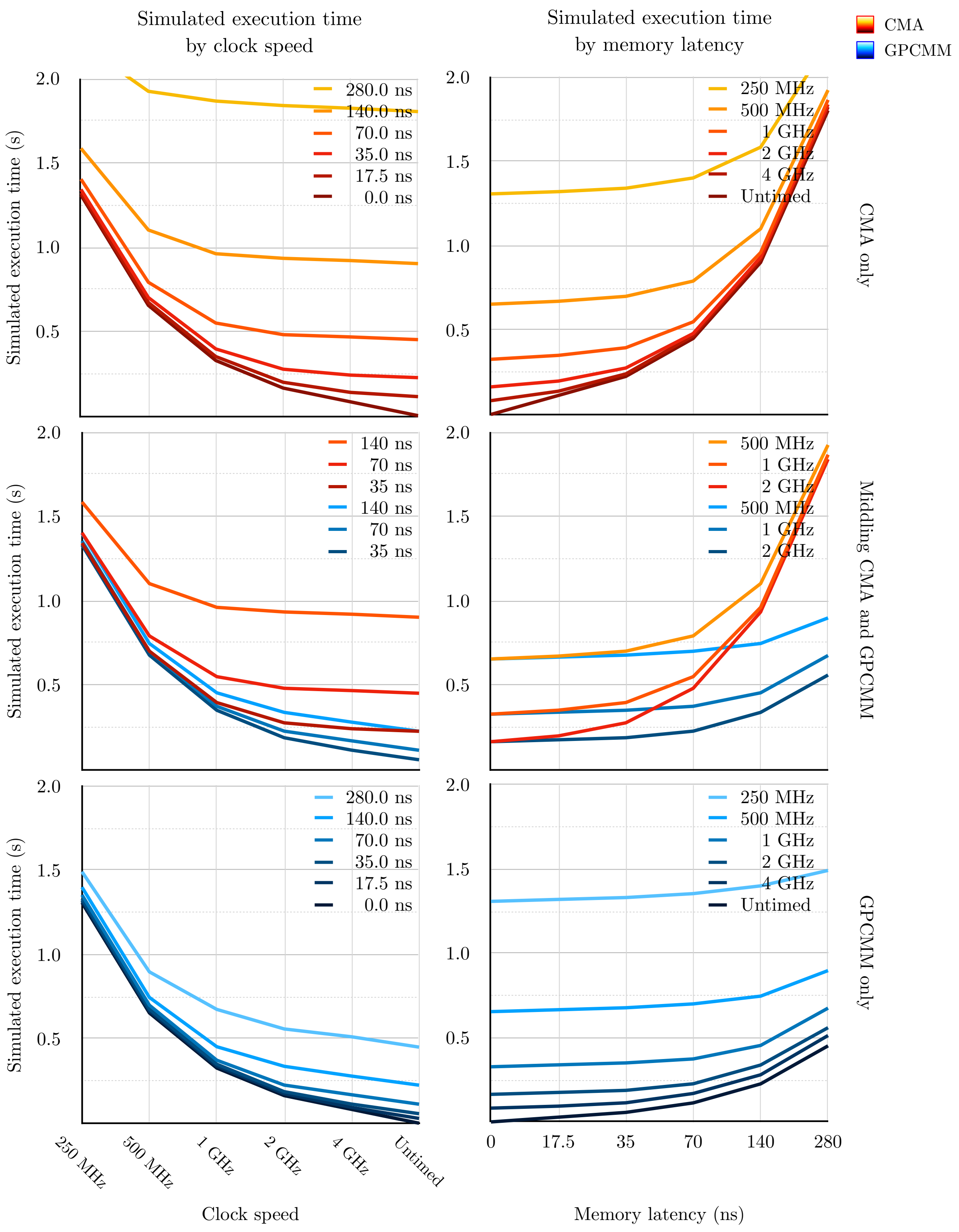

Figure 6.3: Cross-sectional simulated execution times of the CMA and GPCMM by processor speed and by memory latency.

Next, we compare all three of our architecture configurations at the extreme values of no memory latency and untimed processing delay, and at our standard, practical values to verify we are obtaining reasonable results. We take samples from our original data in Table 6.3, Table 6.5, and Table 6.4 and populate Table 6.6, from which we plot Figure 6.4.

For no memory latency, we find that the simulated execution times are the same between all configurations. This is expected, as each configuration has the same application mapped to it and that there is fairness between the experiments. Additionally, this establishes the ideal 0% memory bottleneck baseline. For the untimed processing delay, the GPC/MM configurations perform significantly better than the CMA. This shows that the GPC/MM operate much more effectively even at total memory saturation. Lastly, for the practical parameters of 1 GHz and 2.5, 70 ns[1] memory latency, we see that the GPC is around 66% faster than the CMA. Another important takeaway is that the GPC's simulated execution time is very near its value for no memory latency. Under these practical parameters, the GPC is experiencing almost no memory contention, and is running near the ideal minimum for simulated execution time.

[1]The 2.5 ns on-chip memory latency only applies to the GPC. The other configurations remain at 70 ns.

Table 6.6: Simulated execution time in seconds for all three configurations at extreme and practical values (see footnote 1).

	CMA	**GPCMM**	**GPC**
1 GHz 0 ns	0.33	0.33	0.33
Untimed 2.5, 70 ns	0.45	0.11	0.09
1 GHz 2.5, 70 ns	0.55	0.37	0.33

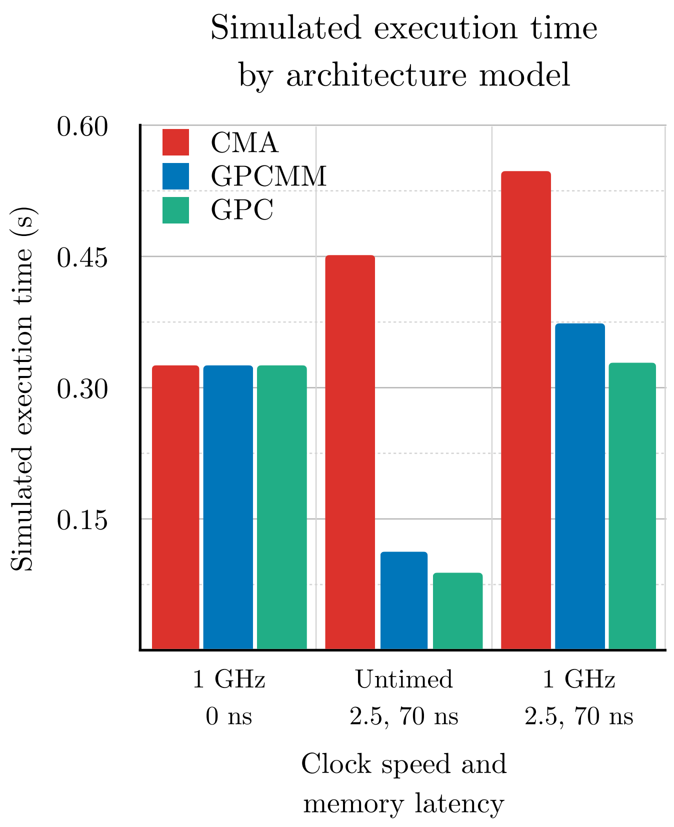

Figure 6.4: Comparison of the architecture configurations at extreme and practical values (see footnote 1).

Chapter 7

Conclusion

In this work, we have introduced a checkerboard grid of processing cells architecture, with distributed processors, memories, and memory controllers, designed to maximize the benefits of parallelization. We had two goals: first, to develop a robust platform with which to build the checkerboard and a classical multi-core model, and second, to verify and compare these models' behavior. We proved to be successful on both fronts.

Our platform proved to be robust in that we easily mapped a JPEG encoder application onto the processing cores for both the checkerboard and classical architecture, varied the speed and timing parameters of their component modules for our experiments, and yielded expected results. The experimental data shows that the checkerboard was generally faster in its execution than the classical multi-core architecture under the same conditions, and it suffered less from slower processor and memory speeds. At the practical parameters of 1 GHz processor clock speed, 2.5 ns on-chip memory latency, and 70 ns off-chip memory latency, the checkerboard's simulated execution time was 66% faster, and experienced almost no memory contention.

7.1 Future work

While these models were explored at a high level, the work presented here provides a foundation for deeper research into the grand challenge of hardware bandwidth bottlenecks. Further work with the current code base for the simulation platform may include exploring different applications, and checkerboard sizes. A useful addition would be a tool to visualize the utilization of the processors, memories, and controllers throughout the simulation. Diving deeper, future work should concern moving from Loosely-Timed to Approximately-Timed TLM and fleshing out implementation details we omitted such as caching and more accurate burst mode memory. The final form of this platform would ideally emulate a full instruction set such as RISC-V [19].

Bibliography

[1] IBM, "Power4: The First Multi-core, 1GHz Processor." [Online]. Available: https://www.ibm.com/ibm/history/ibm100/us/en/icons/power4/

[2] J. Corbet, A. Rubini, and G. Kroah-Hartman, "Direct Memory Access," in *Linux Device Drivers*, 3rd ed. O'Reilly Media, Feb. 2005, pp. 440–442.

[3] H. Gilbert, "The GPU (Specialized Processors)," *Yale University*, Feb. 2010. [Online]. Available: https://pclt.sites.yale.edu/gpu-specialized-processors

[4] D. Efnusheva, A. Cholakoska, and A. Tentov, "A Survey of Different Approaches for Overcoming the Processor-Memory Bottleneck," *AIRCC Journal of Computer Science and Information Technology*, vol. 9, pp. 151–163, 2017.

[5] R. Dömer, "Grid of Processing Cells (GPC)," Mar. 2021, published: Personal communication.

[6] R. Eigenmann and D. J. Lilja, "Von Neumann Computers," Jan. 1998.

[7] R. Merritt, "CPU designers debate multi-core future," *EE Times*, Feb. 2008. [Online]. Available: https://www.eetimes.com/cpu-designers-debate-multi-core-future/#

[8] S. Swan and Cadence Design Systems, "An Introduction to System Level Modeling in SystemC 2.0," Open SystemC Initiative (OSCI), Tech. Rep., May 2001. [Online]. Available: https://newport.eecs.uci.edu/~doemer/eee_uci_edu/09f/18415/resources/WhitePaper20.pdf

[9] IEEE, "Ieee standard for standard systemc language reference manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, 2012.

[10] The European Space Agency, "System-Level Modeling in SystemC." [Online]. Available: https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Microelectronics/System-Level_Modeling_in_SystemC

[11] International Telecommuncation Union, "Digital Compression and Coding of Continuous-tone Still Images," Tech. Rep. T.81, 1993.

[12] Devcore, "DCT-8x8.png," 2012. [Online]. Available: https://commons.wikimedia.org/wiki/File:DCT-8x8.png

[13] D. A. Huffman, “A Method for the Construction of Minimum-Redundancy Codes,” *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[14] A. Moffat, “Huffman Coding,” *ACM Comput. Surv.*, vol. 52, no. 4, Aug. 2019, place: New York, NY, USA Publisher: Association for Computing Machinery. [Online]. Available: https://doi.org/10.1145/3342555

[15] E. Malekzadeh and R. Dömer, “Fast Loosely-Timed System Models with Accurate Memory Contention,” 2022.

[16] W. Chen, *Out-of-order Parallel Discrete Event Simulation for Electronic System-level Design.* Springer International Publishing, 2015. [Online]. Available: https://doi.org/10.1007/978-3-319-08753-5

[17] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[18] Texas Instruments, “CDx4HC405x, CDx4HCT405x High-Speed CMOS Logic Analog Multiplexers and Demultiplexers,” Tech. Rep., 2019. [Online]. Available: https://www.ti.com/lit/ds/symlink/cd74hc4052.pdf

[19] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, “The risc-v instruction set manual, volume i: User-level isa, version 2.0,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54, May 2014. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

Appendix A

Definitions

Table A.1: Definitions for initialisms and acronyms.

GPC	Grid of Processing Cells. Generally refers to the checkerboard.
CMA	Classical Multi-core Architecture.
GPCMM	GPC Memory-matched.
TLM	Transaction-level modeling. A SystemC standard.
FIFO	First in, first out. I.e., a normal queue.
RGB	Red, green, blue color channels.
YCC	YCbCr. Grayscale luminosity and red-blue chromaticity, relative to green.