MapGL: Interactive Application Mapping and Profiling on a Grid of Processing Cells

Claudio Raccomandato
Politecnico di Torino
Turin, Italy
claudio.raccomandato@studenti.polito.it

Emad M. Arasteh, and Rainer Dömer
CECS, University of California, Irvine
Irvine, USA
{emalekza,doemer}@uci.edu

***Abstract*—The Grid of Processing Cells (GPC) has been proposed as a scalable many-core architecture, modeled using SystemC TLM-2.0 methodology. This work introduces a graphical CAD software called Map Grid-based Layouts (MapGL) to facilitate the design process of GPC-based applications, automatically generate their SystemC models, and perform analyses on memory usage and speed. Using MapGL, we map a GoogLeNet Convolutional Neural Network (CNN) to a suitable GPC and improve it with a new modular Memory Access Resources and Interfaces (MARI) library for better communication between processing cells and lower resource usage.**

***Index Terms*—System modeling, SystemC-TLM2.0, CAD**

I. Introduction

Over the last two decades, computer systems focus shifted from raising the clock frequency toward increasing the number of processors [1]. This trend led to higher design complexity and shared memory contention caused by the "memory wall" problem [2]. The growing complexity drives the need for modeling systems at higher abstraction levels using SystemC to evaluate and optimize them early in the design process. Here, we introduce a graphical CAD tool called *Map Grid-based Layouts (MapGL)* to facilitate the design of embedded many-processor systems, automatically generate SystemC models, and optimize resource usage and speed.

A. Grid of Processing Cells (GPC) Platform

Traditional single-, multi-, and many-core computer architectures suffer from the well-known *memory bottleneck* to a single shared main memory which can delay many-core processors for thousands of cycles due to bus contention despite sophisticated multi-level cache hierarchies [3]. As an alternative *scalable* computer organization, tiled network-on-chip architectures have been proposed with separate local memories, such as the Grid of Processing Cells (GPC) [4] where processor-memory pairs are arranged on-chip in a two-dimensional array with only local interconnect.

The checkerboard variant of a GPC is shown in Fig. 1. Processor cores C_{yx} and local memories M_{yx} are arranged in an alternating pattern so that every processor can access to its four neighboring memories. Instead of a shared bus, the combination of a multiplex and de-multiplex interconnect arbitrates the memory accesses in each cell. Specified as a high-level model in SystemC TLM-2.0 with socket-based interconnect, the checkerboard GPC can serve as a starting point for design space exploration of scalable computing platforms without a shared memory bottleneck.

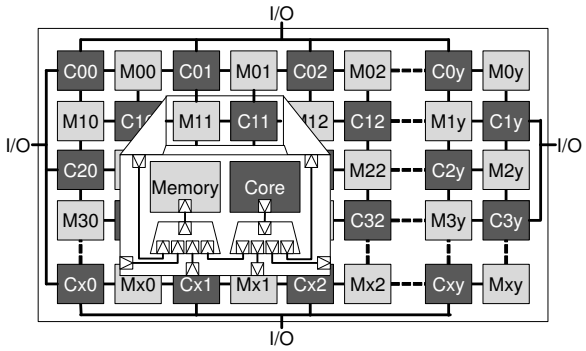


Fig. 1: Checkerboard Grid of Processing Cells (GPC) [4].

One of the main challenges for grid-based architectures is *programmability*. Applications cannot be developed by traditional methods with the assumption of one shared memory. Instead, software must be explicitly partitioned among the cells. Instructions must be mapped to processing cores, and data must be allocated in local memories. Manual partitioning is possible, but tedious and error-prone.

B. Problem Definition

This work aims to achieve three goals:

1) *Demonstrate the scalability and usability of GPC* with two applications of different size-complexity.
2) *Improve the mapping process of GPC applications* through an interactive GUI.
3) *Simplify performance evaluation and inter-cell communication* of GPC-based models

II. Related Work

A large body of research addresses the partition and mapping problem to many-core network-on-chip (NoC) platforms [5], [6]. Yang et al. [5] proposed a multi-application mapping method on the many-core NoC that finds a region on the NoC for each application and then performs a task mapping that maps all tasks of the application into each region. Murali et al. [6] proposed a methodology to map different use-cases onto the NoC architecture, satisfying the performance constraints of each use-case. While these works focus on many-core architecture mapping, our work is a holistic application mapping approach on the grid of processing cells.

Bruch et al. proposed a graphical user interface computer-aided design (CAD) tool which allows the user to evaluate

the performance of NoCs systems using traffic generators in SystemC simulations [7]. While the proposed BrownPepper simulator [7] allows to design and profile RTL and transaction-level models on a 2D-mesh SoC architecture, it does not allow the user to interact, visualize and map an application.

Platform Architect is a commercial software from Synopsys based on SystemC TLM that enables designers to analyze SoC architectures using a graphical user interface [8]. Still, designers are limited to existing features of the proprietary environment, and creating compatible IP models is challenging. MapGL offers free and open-source solutions with more fine-grained controllability over the entire design process and customization for exploring grid-based architectures.

III. Map Grid-based Layouts: MapGL

We present MapGL, an highly configurable CAD software which lets the user design a custom GPC, automatically generate the SystemC model, and analyze it. [9] The user can focus on the application design without distraction due to GPC and programming technicalities. A MapGL design model is highly configurable and portable, saved as a single JSON file. Currently, MapGL works only with the GPC but it can easily support other grid-based many-processors architecture. The cores consist of *Modules*, basic processing elements running C++ code, which can be replaced with SystemC-based processors or accelerators.

Fig. 2 shows the MapGL design flow from the application mapping to the generation of the SystemC model, while Fig. 3 shows MapGL's main window with a Canny Edge Detector [10] example opened.

A. Application Mapping

On the main window, MapGL shows the user-configurable GPC structure in the *Memories View*, where the red squares represent the cores, and the blue ones the memories. An alternative *Channels View* can be selected, allowing the user to focus on cores communication. Inside the MapGL editor, every design is defined using *Modules* and *Channels*. A module describes the behavior of a core using C/C++ code, while a channel allows two or more cores to communicate. The application modules are listed hierarchically on the left side of the window and can be mapped interactively by drag-and-drop to the cores. External I/O interfaces of the GPC are displayed for configuration on the edges of the GPC grid in the channel view. All components and the GPC itself are configured using adjustable parameters on the right side of the window.

B. SystemC TLM-2.0 Project Generator

Once the application mapping has been completed, MapGL validates the platform and automatically generates the SystemC code for the designed GPC application. Users can configure the specific simulation environment for software dependencies via a preferences dialog. The structure of the generated SystemC project is shown at the bottom of Fig. 2. An initial testbench and a `Makefile` are also generated to compile and run the simulation immediately.

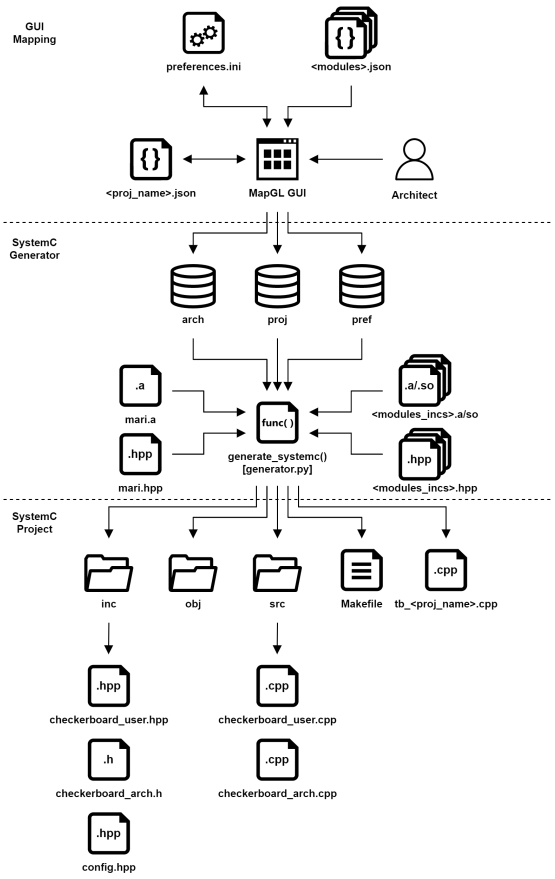
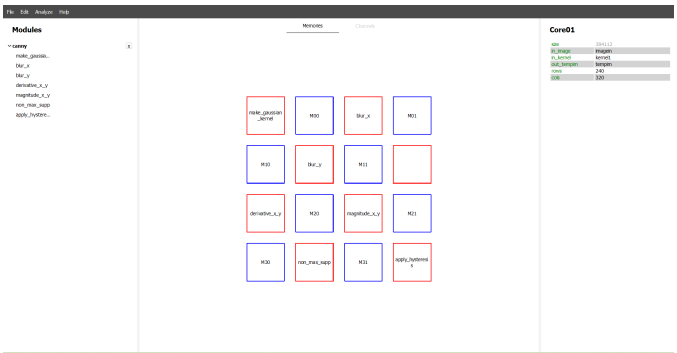
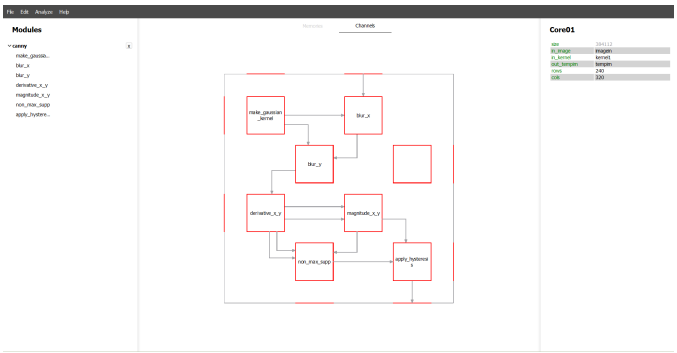


Fig. 2: MapGL design flow and generated files structure.



(a) Memories View



(b) Channels View

Fig. 3: MapGL mapping of a Canny application.

C. Example of a Canny Edge Detector on a 4-by-2 GPC grid

As an example of MapGL usage, we map a Canny Edge Detector [10] application to a 4-by-2 Checkerboard GPC, as shown in Fig. 3. Listing 1 shows an extract of the canny modules package where its modules (e.g. `blur_x`) are described in terms of parameters (e.g. `rows`) and parameters' attributes (e.g. `val`). Each module's actual behavior must be defined in a C++ function template as shown in Listing 2 for the `blur_x`. The references to the *FIFO_out* and *FIFO_in* interfaces (Sec. IV) are required to communicate with the surrounding cores using the `pop()` and `push()` methods.

```
{
  "dependencies" : {
    "SystemC" : {
      "inc_dirs" : ["./inc", "../../3rdparty/mari/inc"],
      "lib_files" : [
        ["./lib", ["libcanny.a"]],
        ["../../3rdparty/mari/lib", ["libmari.a"]]
      ],
      "modules_dec" : ["canny.hpp"]
    }
  },
  "modules" : {
    "blur_x" : {
      "size" : { "val" : 384112, "readonly" : true },
      "in_image" : { "val" : null, "type" : "str" },
      "in_kernel" : { "val" : null, "type" : "str" },
      "out_tempim" : { "val" : null, "type" : "str" },
      "rows" : { "val" : 240, "template" : true },
      "cols" : { "val" : 320, "template" : true }
    },
    ...
  }
}
```

Listing 1: Extract of the Canny modules package (JSON).

```
template<int rows, int cols>
void blur_x(FIFO_out& in_image, FIFO_out& in_kernel,
     FIFO_in& out_tempim) {
  KERNEL kernel;
  IMAGE<cols*rows> image;
  FIMAGE<cols*rows> tempim;
  // getting inputs
  in_kernel.pop(&kernel, sizeof(KERNEL));
  in_image.pop(&image, sizeof(IMAGE<cols*rows>));
  // algorithm execution
  ...
  // generating output
  out_tempim.push(&tempim, sizeof(FIMAGE<cols*rows>));
}
```

Listing 2: Extract of the *blur_x* module implementation.

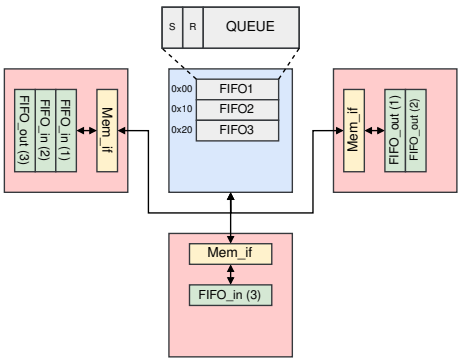


Fig. 4: Example of MARI channels between three cores.

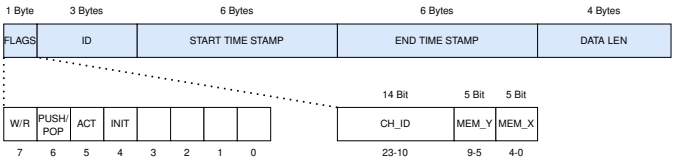


Fig. 5: *mari.log* memory access encoding.

IV. MARI: Memory Access Resources and Interfaces Library

The GPC structure forces each core to communicate by reading and writing data into shared memories. However, a standard communication protocol makes data exchange and memory contention manageable. Our MARI library simplifies interactions and optimizes memory usage by providing a set of software-based FIFO channels that the user can use to transfer data directly between cores.

A. Interfaces

The library is divided into *Memory Interfaces* and *Channel Interfaces*. The memory interface connects a core to *one* memory through TLM-2.0 sockets and provides basic `read()` and `write()` methods to access the memory array. The channel interface represents a second layer of abstraction, e.g. a FIFO interface,[1] which provide interface-specific methods, like `push()` and `pop()`, to the connected memory. The MARI library distinguishes between input (`FIFO_in`) and output interfaces (`FIFO_out`). Fig. 4 shows a simple example of three cores and a shared memory.

A FIFO interface manages its data queue as a ring buffer using two memory-stored counters for sent (S) and received (R) bytes. MARI automatically handles the needed inter-core synchronization through interrupts when the queue is empty and full (blocking communication).

B. Logging for Profiling

During a simulation, MARI can generate a record of all the memory accesses and store it in a file called `mari.log`, which can be used to analyze the traffic and profile the platform. The log file is encoded in binary format to reduce its size, as shown in Fig. 5. Each entry logs the operation performed, the channel and memories used, and the simulation time it started and ended.

V. MapGL Profiling

Assisted by MARI, MapGL allows designers to analyze the memory usage and timing of the application. Memories usage analysis is static, whereas timing analysis is dynamic and requires running a simulation.

A. Memories Usage Analysis

Memory usage is analyzed for each GPC cell based on the space required by the core program and communication channels.The analysis results are available as a numeric report and also graphically as a heat map of the grid cells, as shown in Sec. VI.

[1]Other data structures, e.g. STACK, can be added in the future.

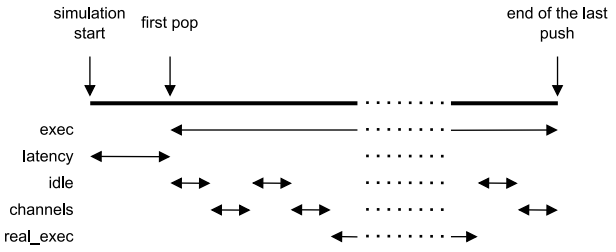


Fig. 6: Core different delays contributions.

B. Timing Analysis

Before simulating to estimate the application's performance, the user can configure memory read and write delays, interconnect propagation delays, and cores computation delays. By exploiting MARI profiling capabilities (`mari.log`), MapGL tracks the timing of the application in detail during a simulation and generates a report about the application speed. For each core, the report distinguishes between five delay contributions:

- The *latency* is the interval from the start of the simulation to the first pop operation.
- The *execution delay* is the span between the first pop and the last push.
- The *idle delay* represents the sum of all the moments in which the core's channels wait to push or pop data.
- The *channels delay* is the sum of the intervals in which the core's channels read or write data to the memories.
- The *real execution delay* represents the actual computation delay of the core, and it is equivalent to the execution delay without idle and channels delay.

Fig. 6 shows how the five delay contributions interchange. The analysis results are also available as a heat map of the grid cells, as shown in Sec. VI.

VI. Experiments and Results

This section will analyze the design process and the experimental results of three SystemC TLM-2.0 models of the GoogLeNet CNN mapped and profiled using MapGL.The first model, called *preliminary*, represents the first attempt to map the application. The other two models use an improved mapping; one tends to increase the application's speed, called *high-speed*, and the other optimizes the memory usage, called *low-memory*.

A. Case study: GoogLeNet CNN

The GoogLeNet is a state-of-the-art CNN for image classification, winner of the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) 2014 with only 6.67% top-5 error [12]. The network comprises 22 layers when counting only layers with parameters or 142 if not. Fig. 7 shows the entire structure of the GoogLeNet CNN. A SystemC model of the GoogLeNet CNN that uses the Caffe model [13], and OpenCV [14] has been designed [15], [16]. This work will use that model as a reference for the new ones mapped on the GPC architecture.

For the GoogLeNet timing, we analyze the computational complexity of layers in terms of the number of multiplications (N_{mul}) and the number of additions (N_{add}). The size of the input volume to each layer is $W_i \times H_i \times C_i$ where W_i, H_i and C_i represent the width, height, and a number of channels, respectively. The most computationally expensive layer in GoogLeNet, the convolution layer, has the following hyper-parameters: K number of filters, kernel size of F, stride S, and padding P. The total number of weights in a convolution layer is $F{\cdot}F{\cdot}C_i{\cdot}K$, and the total number of biases is K. To compute all output elements for all K filters, total number of required multiplications is $N_{mul} \approx \frac{W_i \cdot H_i \cdot Ci \cdot F^2 \cdot K}{S^2}$ and total number of additions is $N_{add} \approx \frac{W_i \cdot H_i \cdot Ci \cdot F^2 \cdot K}{S^2}$. The computational complexity of other constituent layers of GoogLeNet, such as pooling, rectifier, etc., is analyzed similarly in terms of the total number of multiplications and additions [15].

Given a 32-bit single-precision floating-point multiply-accumulate (FP32-MAC) unit available, we assume the total computational latency of a layer to be the product of the number of MAC operations and the inverse of the peak floating-point operations per second (FLOPS): $N_{MAC} \cdot \frac{s}{flop}$. The peak FLOPS value is the maximum number of single-precision floating-point MAC operations each core can perform per second. Hence, computational latency in time units can be evaluated by knowing the core maximum number of FLOPS.

B. GoogLeNet CNN on GPC

The mapping strategy was to exploit the scalability of the GPC architecture by manually assigning one CNN layer per core using as few additional cores as possible. As shown in Fig. 7, the CNN structure can be divided into three parts: the initial nine layers connected in series, the nine inception blocks, and the final four layers in a row. Since most of the network consists of repeating inception blocks, focusing on finding a good mapping for it simplified the overall design problem. Fig. 7 shows how each inception block's first and last layer connects four parallel *paths*: one formed by two layers, one by three layers, and two by four layers. Positioning the last layer next to the first layer of the following inception block avoided the use of additional cores in between inception blocks and increased its modularity.

The inception block mapping of the *preliminary* model, shown in Fig. 8 (a), does not uses any channel abstraction layer, causing each memory to contain only one channel. Additional *Fork* and *Merge* modules were added to serialize and deserialize data in case of multiple initiators or targets, increasing the number of required cores up to 20. The improved mapping, shown in Fig. 8 (b), uses the MARI library to create in the same memory parallel channels of different sizes, which brings back the number of cores to 15.

Fig. 9 shows the two MapGL mappings for *preliminary* model and the *improved* model. The first layers of CNN are shown in yellow, the last ones are in green, and the inception blocks are in red. Four external memories are placed in the north, east, west, and south of the model (not shown in the figure). The north external memory stores the input images, two memories on the sides shorten the path between cores, and the south memory stores the output data.

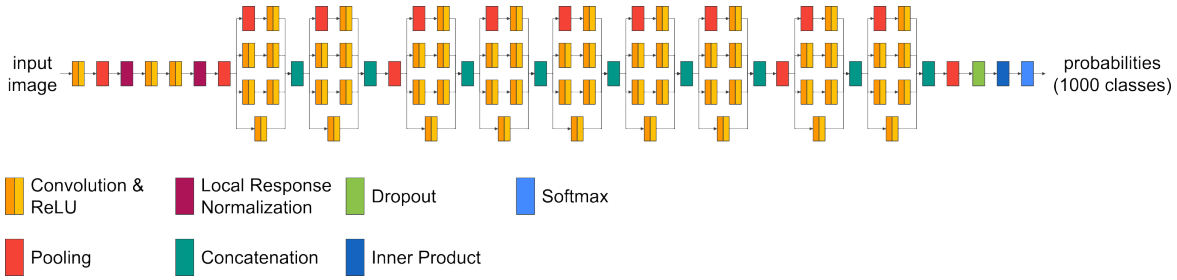
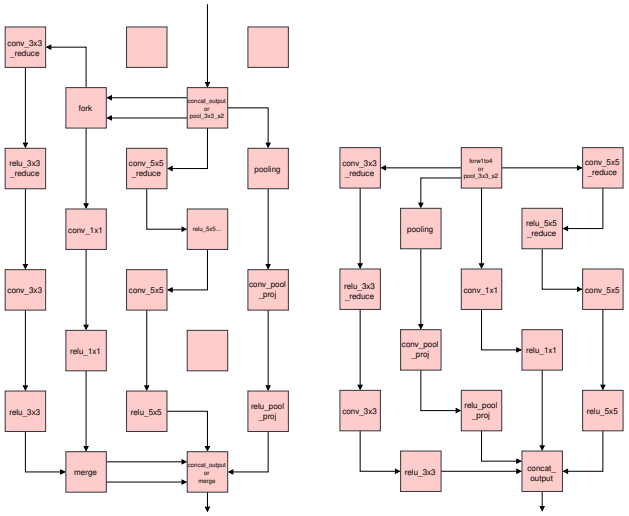


Fig. 7: GoogLeNet CNN structure, redrawn from [11].



(a) Preliminary mapping (b) Improved mapping

Fig. 8: GoogLeNet CNN on GPC inception block design.

The *preliminary* mapping uses a 15×13 grid for a total of 195 available cores and 26 unused. Meanwhile, the *improved* mapping uses a 15×10 grid with 150 available cores and only one unused. Overall, the improved mapping has 7 cores more than the theoretical 142.

For the *preliminary* model and *improved high-speed* model, the size of each FIFO is equal to the size of the payload (if lower than the GPC max memory size) so that the push or pop takes just one or few iterations, reducing the number of stalls. For the *improved low-memory* model, each FIFO is reduced to an arbitrarily small value of *64 bytes*.

For MapGL timing analysis, we rely on the memory read and write delays in [17]. The off-chip memories are assumed to be DRAMs, and the on-chip memories SRAMs. The propagation delay of the multiplexer is arbitrarily chosen to be one-tenth of the on-chip memory read delay, as shown in Tab. I. The Ara vector processor [18] was used as reference for its 16.9 DP-GFLOPS needed to evaluate the computational delay in time units of each core starting from the complexity (Sec. V) as $\frac{N_{add} + N_{mul}}{16.9\ GFLOPS}$ (e.g. first convolution delay $\approx 14\,ms$).

C. Comparison

The memory usage heat maps of the *preliminary*, *improved high-speed*, and *improved low-memory* models are shown in Fig. 10, while the heat maps for execution, communication, and idle delays are shown in Fig. 11, Fig. 12, and Fig. 13.

Looking at Fig. 10, it is clear that the first layers of the GoogLeNet CNN require more memory to store the first transformations of the input image. However, the smaller FIFOs used in the *improved low-memory* model make the memory usage more uniform.

Delay Type	Delay [ns]
Off-chip memory read (DRAM)	50
Off-chip memory write (DRAM)	50
On-chip memory read (SRAM)	2.5
On-chip memory write (SRAM)	2.5
Multiplexer propagation	0.25

TABLE I: Timing analysis communication delays.

There is a correlation between the memory usage in Fig. 10 and the communication delays in Fig. 12 even if the heatmaps' normalization makes this result less evident. Specifically, increasing the size of the FIFOs reduces the channel's delay because it decreases the number of accesses to the shared memory required to read the entire data.

Tab. II shows the simulation results of the three models. The improved mapping reduces by almost one-fourth the grid size compared to the preliminary. This result caused the two following models to become faster and require fewer memories. Overall, the *high-speed* model performs better than the *preliminary* model, while the *low-memory* model represents a valuable alternative to reduce memory consumption.

D. Throughput

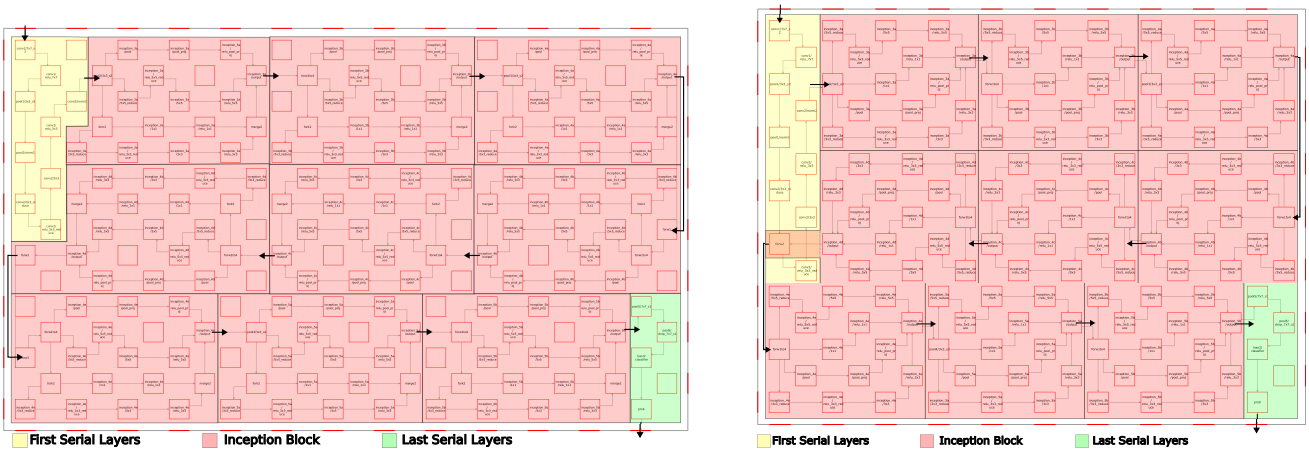
The application's throughput was evaluated after feeding 500 images to the three models. The resulting throughput in each case was around *23 fps*. As shown in Fig. 12, the communication delays of the first layers are much higher than the rest of the structure, which leads to a *bottleneck* that reduces drastically the throughput of the application. One possible solution could be partitioning these layers into multiple cores to perform the operations in parallel, reducing the overall execution delay. A clever design with fine-grain pipelining could increase the calculated throughput.

VII. Conclusion

This paper presented the MapGL editor to map and evaluate the performances of three GPC-based GoogLeNet CNN models, exploiting the grid scalability of the GPC architecture up to 195 cores. All the models were interactively mapped, automatically generated in SystemC, and profiled using the MapGL built-in analysis tools. The MARI library reduced the grid size by one-fourth and improved the timing analysis accuracy, making hidden bottlenecks easier to identify. The results showed the superiority of the *high-speed* model over the *preliminary* model, indicating the *low-memory* model as a

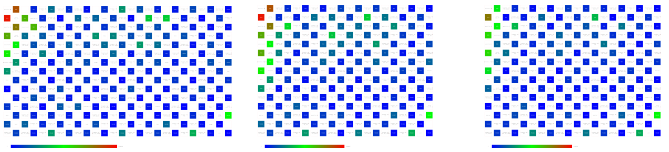
	Grid size	Channels memory usage [kB]	Total memory usage [MB]	Channels delay [ms]	Total delay [ms]
preliminary	15×13	57948	192	92	213
high-speed	15×10	52153	175	82	201
low-memory	15×10	13	123	100	217

TABLE II: Summary of the results of the three models.



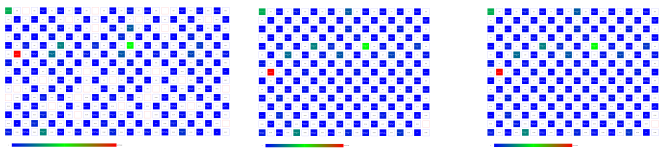
(a) Preliminary mapping (b) Improved mapping

Fig. 9: GoogLeNet CNN on GPC structure.



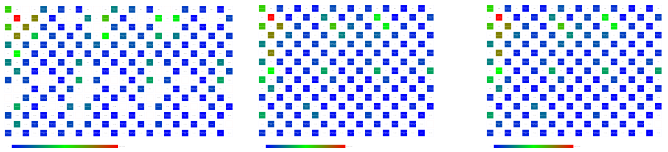
(a) Preliminary (b) High-speed (c) Low-memory

Fig. 10: Memories usage analysis heat maps.



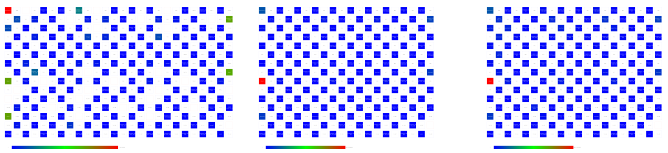
(a) Preliminary (b) High-speed (c) Low-memory

Fig. 11: Timing analysis heat maps, execution delays.



(a) Preliminary (b) High-speed (c) Low-memory

Fig. 12: Timing analysis heat maps, communication delays.



(a) Preliminary (b) High-speed (c) Low-memory

Fig. 13: Timing analysis heat maps, idles delays.

valuable trade-off to reduce memory usage.

References

[1] L. Azriel, A. Mendelson, and U. Weiser, "Peripheral memory: a technique for fighting memory bandwidth bottleneck," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 54–57, 2015.

[2] S. A. McKee, "Reflections on the Memory Wall," in *Proceedings of the 1st Conference on Computing Frontiers*, ser. CF '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 162. [Online]. Available: https://doi.org/10.1145/977091.977115

[3] G. Liu, T. Schmidt, A. Dingankar, D. Kirkpatrick, and R. Dömer, "Optimizing Thread-to-Core Mapping on Manycore Platforms with Distributed Tag Directories," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Jan. 2015.

[4] R. Dömer, "A Grid of Processing Cells (GPC) with Local Memories," Center for Embedded and Cyber-physical Systems, University of California, Irvine, Tech. Rep. CECS-TR-22-01, Apr. 2022.

[5] B. Yang, L. Guang, T. C. Xu, A. W. Yin, T. Säntti, and J. Plosila, "Multi-application multi-step mapping method for many-core network-on-chips," in *NORCHIP 2010*, 2010, pp. 1–6.

[6] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. De Micheli, "Mapping and configuration methods for multi-use-case networks on chips," in *Asia and South Pacific Conference on Design Automation, 2006.*, 2006, pp. 6 pp.–.

[7] J. V. Bruch, M. R. Pizzoni, and C. A. Zeferino, "Brownpepper: A systemc-based simulator for performance evaluation of networks-on-chip," in *2009 17th IFIP International Conference on Very Large Scale Integration (VLSI-SoC)*, 2009, pp. 223–226.

[8] "Synopsys Platform Architect." [Online]. Available: https://www.synopsys.com/verification/virtual-prototyping/platform-architect.html

[9] C. Raccomandato and R. Dömer, "Modeling and Mapping of a GoogLeNet CNN on a Grid of Processing Cells," Center for Embedded and Cyber-physical Systems, University of California, Irvine, Tech. Rep. CECS-TR-23-01, Mar. 2023.

[10] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, 1986.

[11] G. Bortolan, I. Christov, and I. Simova, "Potential of rule-based methods and deep learning architectures for ecg diagnostics," *Diagnostics*, vol. 11, no. 9, 2021. [Online]. Available: https://www.mdpi.com/2075-4418/11/9/1678

[12] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9.

[13] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM International Conference on Multimedia*, ser. MM '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 675–678. [Online]. Available: https://doi.org/10.1145/2647868.2654889

[14] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov, "Realtime computer vision with opencv: Mobile computer-vision technology will soon become as ubiquitous as touch interfaces." *Queue*, vol. 10, no. 4, p. 40–56, apr 2012. [Online]. Available: https://doi.org/10.1145/2181796.2206309

[15] E. M. Arasteh, "Transaction-level modeling of deep neural networks for efficient parallelism and memory accuracy," Ph.D. dissertation, UC Irvine, Irvine, CA, USA, 2022.

[16] E. M. Arasteh and R. Dömer, "Fast loosely-timed deep neural network models with accurate memory contention," *ACM Trans. Embed. Comput. Syst.*, jul 2023, just Accepted. [Online]. Available: https://doi.org/10.1145/3607548

[17] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware Software Interface ARM Edition*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.

[18] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1-ghz+ scalable and energy-efficient risc-v vector processor with multiprecision floating-point support in 22-nm fd-soi," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 530–543, 2020.