GPCC: Grid of Processing Cells Compiler with Pareto-Optimal Design Space Exploration

Yutong Wang, Yanda Li, Rainer Dömer
Center for Embedded and Cyber-Physical Systems, University of California, Irvine, CA, USA
{yutongw5,yandal5,doemer}@uci.edu

Abstract

Programming grid–based system-on-chip architectures remains challenging due to the need for efficient task placement and localized communication. This paper presents the Grid of Processing Cells Compiler (GPCC), an automated compilation flow that maps a task graph extracted from a hierarchical data flow specification onto a two-dimensional grid of processing elements with local memories and neighbor-only communication. GPCC analyzes the application graph, performs task and data partitioning, assigns tasks to grid locations, and generates the needed on-chip and off-chip communication, producing an executable SystemC TLM-2.0 model for simulation and evaluation. GPCC further supports automatic design space exploration of checkerboard grid configurations by varying dimensions and identifying feasible mappings, allowing it to derive Pareto-optimal target architectures. Experimental results on embedded applications and synthetic benchmarks show that GPCC reliably transforms dataflow applications into grid-accurate SystemC models and supports efficient architectural design space exploration.

1 Introduction and Motivation

For high performance, parallel and pipelined applications have taken advantage of two-dimensional grid structures in a variety of configurations. From systolic arrays with only basic operators [1] via course grain reconfigurable arrays (CGRAs) with simple processing elements [2, 3], such architectures have grown the compute capacity of their nodes to network-on-chip (NOC) architectures with fully programmable processor cells [4]. At the same time, processing *in* or *near memory* [5] has recently become an active area of research that overcomes the classic von-Neumann bottleneck [6] by co-locating compute and memory units.

Aligned with these advances, we target in this work a *Grid of Processing Cells (GPC)* architecture, namely R-type checkerboard, where pairs of processors with local memories are arranged in a 2D array on a chip [7], as illustrated in Fig. 1.

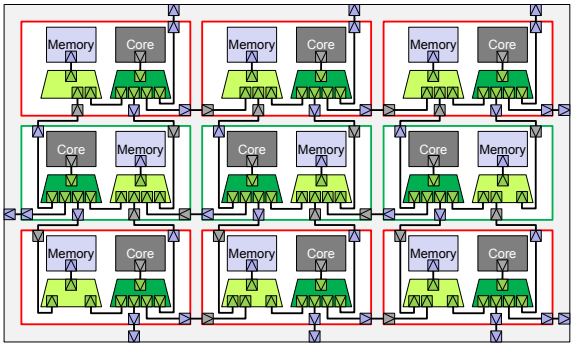

Figure 1: R-type Checkerboard (CBR) in 3x3 size

The main advantage of array architectures is their inherent *scalability* where the height and width of the array can be freely chosen to fit the application at hand. For performance, however, the distance of data movements between cells must be considered. For example, mapping algorithms for NOCs typically minimize the Manhattan distance between communicating cells. Since cells can pass data to any other cell in the network, the number of router hops generally scales linearly ($O(n)$).

In contrast to NoCs, we focus in this work on a *truly scalable* architecture with *constant* communication cost ($O(1)$). We intentionally limit the target checkerboard architecture to communicate only locally. A processor can only reach its own memory and the memories of its immediate neighbors. Specifically, we concentrate on a restricted GPC where processors and memories are arranged in alternating fashion, as shown in Fig. 1. We refer to this target as a R-type checkerboard (CBR), since the cell in the top-left corner has the core on the right side of the memory.

Note that the *true scalability* is an intentional limitation of the architecture and makes the application to architecture mapping harder, but is at the same time a significant performance advantage with a minimal constant 1-hop communication.

Programming a GPC remains a challenge because applications must be partitioned into tasks, mapped onto the grid, and connected through memory channels and I/O ports. Manual mapping is slow and error-prone, while existing heuristic approaches are often rigid and do not generalize well.

The *Grid of Processing Cells Compiler (GPCC)* addresses these challenges by automating the process. GPCC accepts task graph applications specified in C++ or Python and builds an internal representation of modules, ports, channels, and tasks. The task graph model is represented both as a structured object and a NetworkX [8] graph for inspection. The mapping proceeds in two stages. Level-1 mapping places tasks onto cores of a user-specified grid using a recursive search that respects task dependencies and I/O placement. Level-2 mapping adds memory and off-chip I/O assignments and assures the consistency of the result. The mapped system is then translated to SystemC TLM-2.0

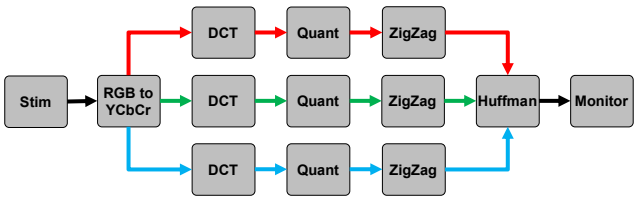

Figure 2: Task Graph of a JPEG Encoder with RGB color channels

[9] where communication channels are realized as memory queues and task operations are refined into SystemC semantics. GPCC also generates a Makefile for direct compilation and, if requested, exports the mapped grid to Graphviz [10] or HTML for easy inspection.

By combining parsing, task graph extraction, mapping, and model generation into a single flow (see Fig. 4), GPCC enables rapid design space exploration (DSE) of checkerboard architectures. Users can adjust grid size, I/O sides, or configuration parameters and immediately obtain a simulatable TLM-2.0 model. In this work, we present the design and implementation of GPCC, explain its mapping algorithms, and demonstrate its effectiveness on both embedded applications and synthetic benchmarks.

1.1 Problem Definition

The checkerboard GPC as shown in Fig. 1, provides a truly scalable platform for parallel computing by arranging processing cores with local memories in a two-dimensional grid with efficient neighbor communication [7]. However, mapping real applications onto a GPC remains a difficult problem. Developers must decide how to partition the application into tasks, assign tasks to specific cells, and route communication through local memories and off-chip I/O. Poor mappings can lead to congestion, load imbalance, or even infeasible designs.

Fig. 2 shows a simple example, the JPEG encoder algorithm, where the application is modeled as a directed acyclic graph (DAG) of image-processing tasks on the three RGB color channels. Each task must be mapped to a core in the checkerboard, and each edge must be assigned to a memory, so that data dependencies are preserved and communication remains efficient. A manual mapping of JPEG encoding onto the CBR grid is shown in Fig. 3, which illustrates the difficulty of balancing tasks, respecting I/O placement, and avoiding communication bottlenecks.

This example illustrates the broader challenge: mapping applications onto a GPC by hand is error-prone and does not scale. Existing heuristic-based methods help, but they remain limited in flexibility and often fail to produce an executable model that can be simulated at the system level. What is needed is an automated approach that extracts the task graph, maps tasks and channels across the grid, and quickly generates a simulatable model for evaluation.

The "forward" task shown in Fig. 3 is not part of the original input task graph. It is introduced only in the manual mapping to simplify the placement process and make the mapping more intuitive for a human designer. The automatic GPCC mapping flow does not insert such auxiliary

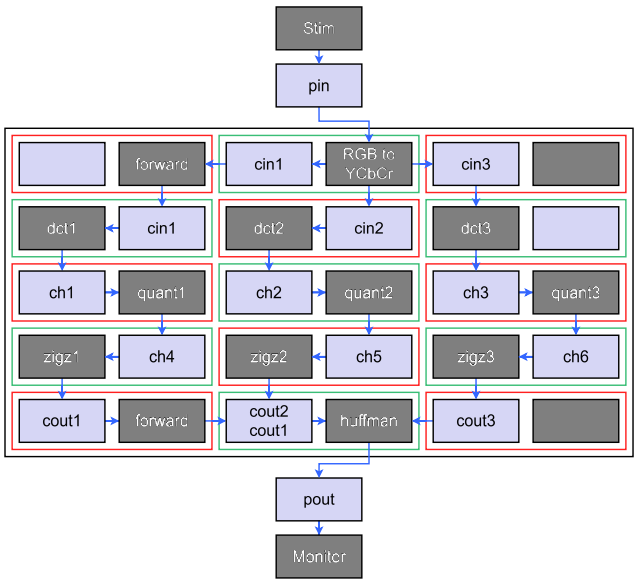

Figure 3: JPEG Encoder manually mapped to a R-type Checkerboard GPC

tasks and operates strictly on the extracted application DAG under a one-task-per-processing-element model. Supporting task-level transformations or allowing multiple tasks to share a processing element remains future work.

1.2 Background and Related Work

Organizing processing elements in regular two-dimensional layouts has been widely explored, for example in coarse-grain reconfigurable arrays (CGRAs) [2, 3], in exploration frameworks such as Canal [11] and CGRA-ME [12], and in tile-based multiprocessor designs such as the Tile Processor Architecture [13] and TILE64 [14]. Grid-like structures also appear in Network-on-Chip (NoC) platforms [4], where routers provide packet-switched communication, e.g., Intel Polaris [15]. While NoC architectures share similarities with our grid layout, they rely on global routing structures, distributed caches, or multi-hop networks. As such, they do not focus on purely local memory access nor strictly neighbor-to-neighbor communication.

In contrast, the Grid of Processing Cells (GPC) [7, 16] enforces a much more constrained and predictable execution substrate: each cell contains its own private local memory, and all communication—both data movement and synchronization—occurs exclusively through direct neighbor cells. There is no global interconnect, shared memory, or long-range routing fabric. This strict locality property is what enables true scalability, since adding more cells does not increase global contention or communication distance.

Existing GPC implementations and studies have so far relied on manual or ad-hoc heuristics tailored to specific applications. Prior works have demonstrated *manual* mappings for Mandelbrot visualization [17], Canny edge detection [18], APNG encoding [19], JPEG encoding [20], and GoogLeNet [21], as well as RTL prototypes and FPGA implementations with RISC-V–based tiles [22, 23, 24]. However, these studies are not compilers in the classic sense:

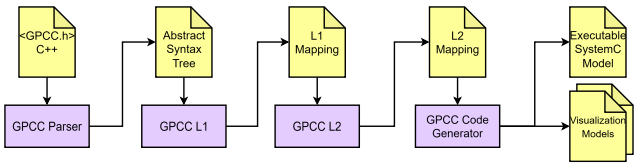

Figure 4: GPCC workflow

each mapping required significant manual effort to balance tasks, assign I/O, choose intermediate memories, and validate communication legality. Moreover, none of these flows automatically produce an executable SystemC TLM-2.0 model following the mapping.

Because of these differences, GPCC is not directly comparable to existing CGRA mappers, NoC compilers, or static placement tools. Those tools typically assume flexible routing, multi-hop communication, or globally shared memory architectures — assumptions that the GPC explicitly does not permit. In this paper, our GPCC instead targets the much stricter R-type checkerboard model, automatically bridging the gap between an untimed task graph and a runnable, grid-accurate TLM-2.0 architecture model. This includes task-graph extraction, task placement, memory and I/O assignment, and generation of complete SystemC code realizing the checkerboard-style neighbor communication semantics.

2 GPC Compiler

GPCC is a command-line compiler that automatically maps an application to a GPC. As shown in Fig. 4, GPCC reads a C++ model and constructs the application's Abstract Syntax Tree (AST). From this input, GPCC extracts the task graph, places tasks on a R-type checkerboard with user-defined dimensions $H \times W$, and generates a corresponding simulatable SystemC TLM-2.0 model that reflects the mapping.

In this section, we first describe how applications are specified, then explain our 2-level mapping algorithm, and finally present the output generation.

2.1 Input Specification

The input application for GPCC is specified as a hierarchical composition of modules (computation) connected by channels (communication). When flattened, the application forms a task graph, as shown in Fig. 2. The system designer can specify the module hierarchy in a C++ file (using a lightweight header `<GPCC.h>`) which the GPCC frontend parses, or by use of a Python API that directly constructs the modules and connecting channels as an internal representation.

Fig. 5 illustrates the GPCC Python API for building the JPEG encoder application. The construction starts with an Application object named `JPEG_Encoder`. Next, a `DCT` component is constructed as a Module object with two ports PortIn and PortOut. The functionality of the `DCT` module is specified as a `main` function that, in an infinite loop, reads a block of image pixels, transforms it using a ChenDCT function, and pushes the result to the output port. Finally, a

```
import GPCC_App
def Construct(AppName="JPEG_Encoder"):
    App = GPCC_App.Application(AppName)
    ...
    m = GPCC_App.Module("DCT")
    m.Ports.append(GPCC_App.Port("PortIn", "QUEUE_IN", "Block"))
    m.Ports.append(GPCC_App.Port("PortOut", "QUEUE_OUT", "Block"))
    m.Main.extend(r"""
        void main(void)
        {   Block in_Block, out_Block;
            while (1)
            {   PortIn.POP(in_Block);
                out_Block = ChenDCT(in_Block)
                PortOut.PUSH(out_Block);
            }
        }""".splitlines ())
    m.Threads.append(GPCC_App.Thread("main", detached=True))
    App.Modules.append(m)
    ...
    m = GPCC_App.Module("Encoder")
    m.Ports.append(GPCC_App.Port("Port1", "QUEUE_IN", "Block"))
    m.Ports.append(GPCC_App.Port("Port2", "QUEUE_OUT", "Block"))
    m.ChannelInsts.append(GPCC_App.ChannelInst("ch1","QUEUE","Block"))
    m.ChannelInsts.append(GPCC_App.ChannelInst("ch2","QUEUE","Block"))
    App.AddModuleInst(m, "dct", "DCT")
    App.AddModuleInst(m, "quan", "Quantize")
    App.AddModuleInst(m, "zigz", "Zigzag")
    m.AddChannelParams("ch1", [FIFO_SIZE])
    m.AddChannelParams("ch2", [FIFO_SIZE])
    m.AddPortMap("dct", ["Port1", "ch1"])
    m.AddPortMap("quan", ["ch1", "ch2"])
    m.AddPortMap("zigz", ["ch2", "Port2"])
    App.Modules.append(m)
    ...
    return App
```

Figure 5: Constructing the JPEG Encoder application with GPCC Python API

Thread is created in DCT to execute the `main` function.

Next, Fig. 5 shows the construction of a hierarchical module `Encoder` that contains an instance of the `DCT` module and two other modules `Quantize` and `Zigzag`. The modules process blocks of image pixels in a pipeline with the help of two channel instances `ch1` and `ch2`.

The construction of the `JPEG_Encoder` application is completed when all modules are instantiated in the App object.

2.2 Level 1 Mapping Algorithm

GPCC applies a two-level mapping strategy. At Level-1, tasks from the application graph are recursively placed onto the cells of the target checkerboard (Alg. 1). For each task, the set of candidate positions is the intersection of the neighbors of already mapped predecessors and successors. If no candidate exists, the search backtracks. In this way, the placement preserves task dependencies while keeping related tasks close together in the grid.

As an example, we illustrate Alg. 1 on the JPEG Encoder application. Fig. 2 shows the task graph with the three color channels, and Fig. 6a shows the result of Level-1 mapping.

The mapping process begins by assigning off-chip modules, such as Stim and Mon, to the off-chip I/O units which serve as source and sink of the input task graph. The DCT successors of the source node, in this case the RGB-to-YCbCr color convertor, are then added to the queue. When RGB-to-YCbCr is dequeued, its candidate positions are computed as the neighbors of the cell assigned to Stim. A valid cell is chosen and the mapping is extended. The same process is repeated for the remaining nodes, with each placement de-