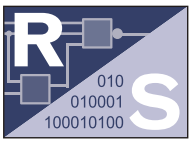
A Coarse-grained Reconfigurable Array for High-Performance Computing

Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
Genehmigte Dissertation im Fachbereich Elektrotechnik und Informationstechnik von M.Sc.
Philipp Sebastian Käsgen
Tag der Einreichung: 31.01.2025, Tag der Prüfung: 30.06.2025

1. Gutachten: Prof. Dr.-Ing. Christian Hochberger
2. Gutachten: Prof. Dr.-Ing. Markus Weinhardt
3. Gutachten: Prof. Dr.-Ing. Li Zhang
Darmstadt, Technische Universität Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT



Electrical Engineering and Information Technology Department
Institut für Datentechnik
FG Rechnersysteme

A Coarse-grained Reconfigurable Array for High-Performance Computing

Accepted doctoral thesis in the department of Electrical Engineering and Information Technology by M.Sc. Philipp Sebastian Käsgen

Date of submission: 31.01.2025
Date of thesis defense: 30.06.2025

Darmstadt, Technische Universität Darmstadt

Bitte zitieren Sie dieses Dokument als:
URN: urn:nbn:de:tuda-tuprints-313693
DOI: https://doi.org/10.26083/tuprints-00031369
Jahr der Veröffentlichung auf TUprints: 2025

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
https://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de

„Siehst Du, Momo", sagte er, "es ist so: Manchmal hat man eine sehr lange Straße vor sich. Man denkt, die ist so schrecklich lang, die kann man niemals schaffen, denkt man." Er blickte eine Weile schweigend vor sich hin, dann fuhr er fort: "Und dann fängt man an, sich zu eilen. Und man eilt sich immer mehr. Jedes Mal, wenn man aufblickt, sieht man, dass es gar nicht weniger wird, was noch vor einem liegt. Und man strengt sich noch mehr an, man kriegt es mit der Angst zu tun, und zum Schluss ist man ganz aus der Puste und kann nicht mehr. Und die Straße liegt immer noch vor einem. So darf man es nicht machen!"

Er dachte einige Zeit nach. Dann sprach er weiter: "Man darf nie an die ganze Straße auf einmal denken, verstehst Du? Man muss nur an den nächsten Schritt denken, den nächsten Atemzug, den nächsten Besenstrich. Und immer wieder nur den nächsten." Wieder hielt er inne und überlegte, ehe er hinzufügte: "Dann macht es Freude; das ist wichtig, dann macht man seine Sache gut. Und so soll es sein.“

Und abermals nach einer langen Pause fuhr er fort: „Auf einmal merkt man, dass man Schritt für Schritt die ganze Straße gemacht hat. Man hat gar nicht gemerkt wie, und man ist nicht außer Puste.“ Er nickte vor sich hin und sagte abschließend. „Das ist wichtig.“

"Momo" (1973), Michael Ende

Zusammenfassung

Hochperformantes Rechnen (HPC) umfasst Applikationen, die rechenintensive Aufgaben auf vielen und verteilten Daten ausführen. Diese müssen auf mehreren Knoten von Serversystemen, die über schnelle Netzwerke kommunizieren, ausgeführt werden, um Ergebnisse so schnell wie möglich zu berechnen. Solche Systeme werden im Zusammenspiel mit den Abbildungen der Programme entworfen, um die Flexibilität, verschiedene Applikationen abbilden zu können, gegen Hardwareressourcennutzung abzuwägen. Trotzdem sind HPC-Anwendungen entweder durch die Speicherbandbreite oder durch die Kommunikationsbandbreite des Netzwerks begrenzt: Einerseits sind die Zugriffszeiten auf Speicher in der Vergangenheit langsamer gesunken als die Leistungsfähigkeit von Prozessoren gestiegen ist, andererseits sind die Latenzen von wachsenden Netzwerken automatisch durch die örtlichen Distanzen eingeschränkt.

Der Entwurf von HPC-Systemen und das Abstimmen der Anwendungsabbildungen auf die höchste Rechenleistung gehen auf Kosten der Leistungsaufnahme: Einen Großteil der Leistungsaufnahme macht die Kühlung der Prozessoren (CPUs) aus. Es ist in der Zukunft wichtig, die Rechenleistung leistungseffizienter zu erreichen. Hardware-Beschleuniger wie grobgranulare rekonfigurierbare Architekturen (CGRAs) sind in dieser Hinsicht vielversprechend, da sie in der Vergangenheit eine bessere Energieeffizienz gezeigt haben als herkömmliche CPUs. CGRAs bestehen aus mehreren Prozessorelementen (PEs), die zur Laufzeit rekonfiguriert werden können, wodurch mehr Abbildungsmöglichkeiten von Anwendungen für energieeffiziente Ausführungen geboten werden.

In dieser Arbeit wird der hochperformante rekonfigurierbare Prozessor (HiPReP) vorgestellt, welcher ein CGRA für HPC ist. Zudem wird eine Methodik präsentiert, um das CGRA so zu parametrieren, dass es die verfügbare Speicherbandbreite während einer Anwendungsausführung - z.B. Matrix-Matrix-Multiplikation (MMM) - nutzen kann, und um die Anzahl der PEs gegen die Anzahl der HiPReP-Instanzen im System abzuwägen. Jede HiPReP-Instanz greift auf den Speicher mittels einer Hardware-Komponente namens Frontend (FE) zu, welche je mit einem Level-1-Cache (L1) verbunden ist. Da das schnelle

Laden und Speichern von Daten essentiell ist, um die Wartezeiten zu minimieren, spielt das FE eine wichtige Rolle für die Rechenleistung. Es wird gezeigt, dass für das gewählte System CGRAs mit 5×5 PEs ausreichen, um die Datenbandbreite eines L1-Caches bei Ausführung einer MMM auszunutzen, und dass darüber hinaus sich mehrere HiPReP-Instanzen besser skalieren lassen. Für andere Anwendungen, die zukünftig auf den HiPReP abgebildet werden, ist ein 5×5-HiPReP ein Ausgangspunkt für weitergehende Optimierungen.

Abstract

High-performance computing (HPC) comprises applications which run compute-intensive tasks on vast and at times scattered data and which need to be run on multi-node server systems communicating over high-speed networks to compute results as fast as possible. These systems are hardware-software-codesigned to guarantee a good trade-off between flexibility for different application mappings and high hardware utilization. Yet, the HPC applications are ultimately memory-bound, on the one hand, or communication-bound, on the other hand: The access latencies of memories historically have not decreased as fast as the compute performance of processors has increased, and the latencies of growing networks are inherently bottlenecking due to the physical distance of the nodes.

Designing HPC systems and tuning application mappings towards the best performance comes at the cost of a high power consumption: A huge portion of the total power goes into cooling the central processing units (CPUs). It is crucial to reduce the power consumption of processors to implicitly perform computations more power-efficiently. Hardware accelerators like coarse-grained reconfigurable arrays (CGRAs) are promising in this respect, since they have shown to have a better energy efficiency than traditional processors in the past. CGRAs feature multiple processing elements (PEs) which can be reconfigured at runtime which opens up more freedom for application mappings and opportunities for efficient execution of tasks.

In this work, the High-Performance Reconfigurable Processor (HiPReP) is proposed, a CGRA designed for HPC. Further, a methodology is presented on how to parametrize this CGRA such that it can harness the available bandwidth when running an application, e.g. a dense matrix-matrix multiplication (MMM), and on how to gauge trade-offs of scaling the number of PEs versus scaling the number of HiPReP instances in a system. Each HiPReP instance accesses the memory through a hardware component which we call frontend (FE) and which is connected to a regular level one cache (L1). Since loading and storing data as fast as possible is paramount to reduce idle times, the FE plays an important role for the achieved performance. We show that for the chosen compute system CGRAs with 5×5

PEs are sufficient to utilize the L1's bandwidth when running MMMs, and that beyond that size scaling up the number of HiPRePs scales better. For other applications which will be mapped in the future, 5×5 CGRAs are a minimal setup to explore the design space.

Contents

1. Introduction

Since the end of the Dennard scaling [1], the clock frequency of central processing units (CPUs) could not be as easily increased to gain more compute performance due to the generated heat. New ways of increasing the compute performance were needed. The most common way so far has been to increase the number of compute cores of multi-core systems.

Esmaeilzadeh et al. [2] predicted that scaling CPU cores will be power limited at some point, too. As a consequence, hardware accelerators are explored. Hardware accelerators are hardware components specialized to certain types of tasks and mostly require a host CPU to offload tasks to them, i.e. they cannot operate standalone. They are designed to perform this task more energy efficiently than the host CPU. In the high-performance computing (HPC) community, for instance, graphic processing units (GPUs) are successfully used as hardware accelerators. Still, GPUs are not suited for every task due to their architecture which will be further explained in section 2.1.

To further increase the compute performance, other accelerator types need to be explored.

1.1. Motivation

In the past, coarse-grained reconfigurable arrays (CGRAs) showed promising results in terms of energy-efficient computing [3], [4], [5], [6]. CGRAs are hardware accelerators which have not yet been widely adapted by the HPC community [7] although reconfigurable chips are being used in data centers like Xilinx' field programmable gate arrays (FPGAs) in Amazon's web services (AWS) [8], and gaining traction for accelerating artificial intelligence like SambaNova's Reconfigurable Dataflow Architecture [9]. Hence, CGRAs are promising candidates to accelerate HPC applications.

When looking at the CGRA landscape in this context, a research gap can be identified, based on the following observations:

1. Application:

    a) Floating point support: As we will see in section 2.2.1, the majority of CGRAs do not feature floating point units (FPUs).

    b) Memory access patterns: HPC applications often access the memory in irregular patterns. These patterns are likely neither spatially nor temporally local which reduces the efficiency of hardware caches. This has not been addressed by the majority of the CGRAs, according to Liu et al. [10].

2. Hardware: Host system integration: CGRAs fully integrated in host systems are rare, thus, making it mostly impossible to gauge their fitness for HPC applications. The aptitude for HPC needs to be assessed in a realistic hardware environment rather than in an isolated experimental setup. This requires at least a realistic model of a memory hierarchy and related access delays to cover HPC criteria, as explained in section 2.3.1. In addition, the system integration has implications on the application mapping (e.g. by a compiler and/or programmer), since, for instance, trade-offs have to be made between when the host processor should do a task or when it is beneficial to hand it over to the CGRA.

For an application to be used in experiments assessing the HPC capabilities of CGRAs, at least one of the application criteria, i.e. floating-point support or irregular memory access patterns, have to be covered.

These considerations have not been covered conclusively by related work, as we will see in section 2.2, and leave a research gap. To evaluate a CGRA in the context of HPC, this gap must be addressed in a single study.

1.2. Contribution of this work

This work explores the HPC capabilities of the HiPReP CGRA. HiPReP's processing elements (PEs) contain an FPU each and are supplied with data from memory through a programmable unit. The memory access patterns used in this work are fairly regular but can be explored with future hardware extensions which are pointed out in the future work section.

To demonstrate the general feasibility, the HiPReP is integrated into a RISC-V [11] based host system with a memory hierarchy and host processor. RISC-V is an open instruction set architecture (ISA) and has been being adapted by the HPC community [12].

The experiments exemplarily show

- how to evaluate the benefit of using the HiPReP over general purpose processors,
- how to make optimal use of the available memory bandwidth,
- how the HiPReP concept scales in terms of number of PEs per HiPReP instance and across multiple HiPReP instances in the same system, as well as
- preliminary performance numbers from compiler mapped benchmarks.

2. Background

For understanding the contribution of this work, first the general principle of CGRAs is explained in section 2.1. Then, the state of the art is covered in section 2.2, before the methodology of analyzing and optimizing HPC accelerators is presented in section 2.3.

2.1. Basics of coarse-grained reconfigurable architectures

CGRAs have been described in the past, e.g. by de Sutter et al. [13], by Wijtvliet et al. [14], or by Li et al. [15]. A CGRA consists of a one- or two-dimensional array of PEs. These PEs are programmable and can execute integer operations, in general, but can perform other operations as well, e.g. handling control flow. Further, some CGRAs support floating point operations. The supply of PE instructions, mostly called contexts, comes from a central context memory for all PEs or from local context memories per PE. In-between the PEs, a network exists for exchanging data. The topology of the network is one major design decision and is typically a bus, mesh, ring, point-to-point, or even a combination of these. Input and output data can be requested from memory mostly by a subset of PEs. For instance, only the outer PEs of the CGRA might generate memory requests. This means that CGRAs commonly have wider memory interfaces than general-purpose CPUs to process data in parallel as much as possible. In short, CGRAs process multiple data streams through their PEs with multiple instructions/contexts.

To position this architecture paradigm in the compute architecture landscape, we can apply Flynn's taxonomy [16], [17]. It classifies compute architectures based on the number of instruction and data streams they can handle. For instance, CPUs can handle one instruction stream and one data stream, and are classified as *SISD* architecture - *single instruction stream, single data stream*. In table 2.1, examples for each of the four classes - *SISD*, *SIMD*, *MISD*, and *MIMD* - are given. The prefix *SI-* stands for *single instruction*

Table 2.1.: Flynn's taxonomy [18]

		# data streams	
		-SD	-MD
# instruction streams	SI-	single-core CPUs	Vector processors, GPUs
	MI-	systolic arrays	multi-core CPUs, *CGRAs*

stream, and *MI-* for *multiple instruction stream*. The postfix *-SD* means *single data stream*, and *-MD* means *multiple data stream*.

Because of these traits, CGRAs are suited for MIMD mappings, but could also service any of the other three categories, even though they might not be optimal for those. In summary, CGRAs offer a lot of flexibility and mapping possibilities which GPUs - the most commonly used HPC accelerator type as of today - are conceptually incapable of.

2.2. Related Work

2.2.1. CGRAs

To explain the concepts of other CGRAs and make the distinction clear between them and this approach, table 2.2 is given with relevant characteristics, building on the criteria from section 1.1. The CGRAs are selected based on whether they are integrated in host systems with main memory and host processor and whether they seem capable to run realistic benchmarks. These characteristics comprise:

a) Host coupling The coupling to the host is either tight or loose. We define *tight* as the CGRA sharing some resource from the host processor's pipeline. In figure 2.1a, the CGRA needs the host's load-store unit (LSU) to provide the data to process, for example. We divide *loose* into two categories, based on whether the CGRA accesses data on a scratchpad memory, or directly from the same main memory as the host. An example for the scratchpad memory category is a GPU which requires a DMA transfer to copy the data to and from the GPU board's memory, as depicted in figure 2.1b. The direct approach is illustrated in figure 2.1c, and requires no support from the host processor other than a task descriptor about what to do.

Table 2.2.: HPC relevant characteristics and their occurrence in the CGRA architectures related to HiPReP

CGRA	Host coupling			separated address generation	Memory access pattern		FP
	tight	loose			indirect	dynamic	
		scratchpad memory	Direct				
DySER	✓				✓	✓	✓
ADRES	✓				✓	✓	
SGMF		✓			✓	✓	✓
Plasticine		✓		✓			✓
T-CGRA			✓[1]	✓	✓	✓	✓
RSVP			✓	✓			
AMIDAR			✓[2]		✓	✓	✓
HiPReP			✓	✓	✓[1]	✓[1]	✓

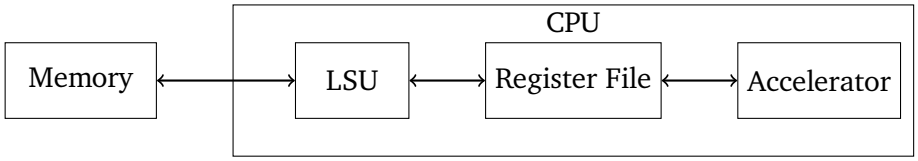
b) Memory access pattern Memory access patterns describe in which sequence addresses are generated to access the memory and are a property of the applications. They can be categorized as direct, indirect, and dynamic. The algorithms 1, 3, and 4, are example code snippets for these different memory access patterns. In algorithm 2, a variation of the direct access pattern is given which is commonly found in stencil codes and matrix operations. In a direct pattern, the address computation does not depend on other load instructions. To better understand what constitutes a dependence on a load, we can look at the other types: In the indirect case, at least one other load is needed to compute the address of the final memory access; in the dynamic case, at least one other load is needed to compute the condition under which the final memory access is executed or not.

paragraphc) Separated address generation This expresses the capability to generate addresses in a separated unit, and not involving the compute units. For the direct and indirect memory access pattern, this separation exposes potentially more memory-level parallelism (MLP). Addresses generated by dynamic memory access patterns do not benefit from this design choice, though.

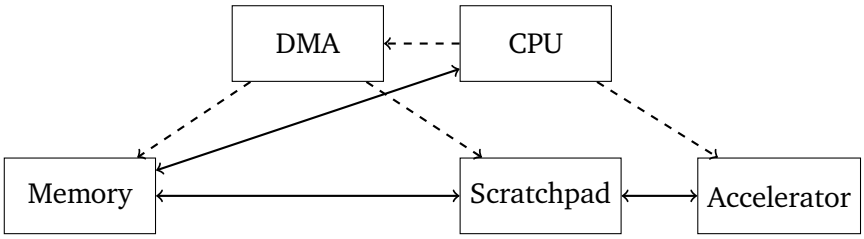
d) FP FP is an abbreviation for floating point operation support.

In the remainder of this section, other CGRAs are analyzed in more detail.

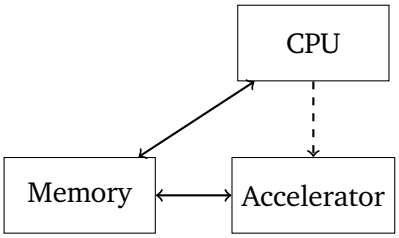
[1] future work, respectively
[2] Standalone system



(a) Tight



(b) Loose and scratchpad memory



(c) Loose and direct

Figure 2.1.: Integration categories

Algorithm 1 Direct memory access pattern

```
for (int i = 0; i < N; i++) {
        sum += input[i];
}
```

Algorithm 2 Direct, nested memory access pattern

```
for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
                sum += input[i][j];
        }
}
```

Algorithm 3 Indirect memory access pattern

```
for (int i = 0; i < N; i++) {
        sum += input[ indices[i] ];
}
```

Algorithm 4 Dynamic memory access pattern

```
for (int i = 0; i < N; i++) {
        int j = (a[i] < b[i]) ? i : N - i;
        c[i] = input[j];
}
```

The **DySER** (Dynamically Specializing Execution Resources) architecture [19], [20] is a CGRA tightly integrated into the pipeline of an OpenSPARC processor. It can access the host's register file, and additionally communicate with memory via a wide memory interface, similar to Intel's Streaming SIMD Extensions (SSE), to consume and produce data. Since the contexts are coming down the processor pipeline as custom instructions and the scalar register file is shared with the host processor, the compiler can generate all variations of memory access patterns. Yet, for the wide memory interface, the addresses of the memory request have to be contiguous which is up to the compiler to identify and exploit after transforming and aligning eligible accesses [21]. The authors say that DySER "speeds up the program through concurrently active functional units, and is limited by the data delivery rate of the integrated processor"[22, p. 1]. In other words, to go beyond the data delivery rate of the integrated processor, loosely coupled CGRAs might have more opportunity to fully utilize the memory bandwidth when they can generate memory requests autonomously and highly parallel.

Mei et al. present the CGRA template **ADRES** (Architecture for Dynamically Reconfigurable Embedded Systems) [23], [24] which is tightly integrated with a Very Long Instruction Word processor (VLIW) by sharing the host processor's register file and functional units. Depending on how the template is set up, more or less data can be requested by the CGRA and used through the register file. The template supports any mesh topology, although the authors mainly explored nearest neighbor communication of up to eight connections. ADRES is one of the two related CGRAs we present which does not support floating point operations, but is note worthy due to its mapping algorithm which is informed by their Modulo Routing Resource Graph (MRRG) abstraction.

As a NVidia CUDA [25] compatible architecture, the Single-Graph Multiple Flows architecture (**SGMF**) [26] features a loosely coupled CGRA aiming at better energy efficiency than GPUs. In its core, a folded hypercube topology connects control units with compute units, such as ALUs and FPUs, as well as with special compute units, such as floating point dividers. These units are surrounded by LDST units (load-store units) on the perimeter. The folded hypercube's switches are connected to the four closest switches and the four closest units each, with a hop link to the second nearest neighbors when the respective switch is on the perimeter. Due to CUDA, they support all memory access patterns and access data on a GDDR5 DRAM. The authors report significant energy savings compared with NVidia's Fermi architecture.

Plasticine [27] demands that data is first transferred to distributed on-chip scratchpad memories. These are part of so-called pattern memory units (PMUs). The transfer of data between the off-chip DRAM and the scratchpad memory is handled by address generators (AGs) with scatter/gather capability. The AGs are connected to the perimeter switches of a mesh network and each of these is connected to the four nearest switches and four pattern units: Two PMUs and two pattern compute units (PCUs). The PCUs operate on four SIMD lanes each and are capable of floating point operations. The read and write accesses to the PMUs' scratchpad memorys are programmed through counters, which enable variations of direct memory access patterns. The authors compare Plasticine with respective FPGA implementations (Intel Stratix V) of several benchmarks but do not give absolute performance numbers. For general matrix-matrix multiplication (GEMM), they state a speedup of 33x, for instance.

Although we define tight coupling differently than Abbaszadeh et al. [28], they propose the Tightly-coupled CGRA (**T-CGRA**). T-CGRA owes its name to the direct access to the L1D cache of the host system, contrary to us who define this as loose. Supporting all relevant memory access patterns, a Data Streaming Unit (DSU) orchestrates these accesses and assures the flow of data to and from the CGRA. The T-CGRA's PEs can operate on floating point numbers and are connected through a 4-nearest-neighbor (4NN) mesh.

The Reconfigurable Streaming Vector Processor (**RSVP**) [29] presents a concept to generate direct memory requests, and to stream data to and from a non-specified compute unit. The memory accesses are integrated in instruction flow with annotations from the compiler when to issue the next memory request descriptor. RSVP's address generating units (AGUs) inspired HiPReP's AGUs but its design does not handle out-of-order responses from memory and, hence, cannot be connected to caches as such.

The Adaptive Microinstruction Driven Architecture (**AMIDAR**) [30], [31] is a reconfigurable processor class operating on Java bytecode. Instructions are first sent to a token

Table 2.3.: NVidia GH100

Feature	value	unit
Streaming Multiprocessor (SM)	144	SM
Tensor Core (TC) per SM	4	TC/SM
FP32 units per Tensor Core	32	FP32/TC
clock frequency	1980	MHz
peak performance (FP32)	51.2	TFlops

generator which then distributes the instructions with a set of token through a distribution network to the required functional units. Data is sent over a dedicated data bus. One of the functional units can be a CGRA which features floating point units and can generate memory requests autonomously and is therefor loosely coupled. Yet, this CGRA is not strictly tied to the AMIDAR architecture and could be used in other systems as well. A prefetching mechanism allows the CGRA to increase the effective memory bandwidth.

2.2.2. Graphics processing units

GPUs were initially used to solely accelerate graphics application like games, but the HPC community recognized the computational potential and has been using them ever since. GPU vendors like NVidia or AMD reacted, and integrated tensor cores, for instance, to accommodate the computational needs of the HPC community - e.g. for machine learning applications.

Since GPUs are established HPC accelerators, we take a look at the performance of a contemporary GPU, NVidia's Hopper architecture H100 [32] to define a point of reference for the contribution of this work in the HPC domain.

The H100 comes in a regular and a full version. The full version of the H100 is called GH100. In figure 2.2, a diagram of a streaming multiprocessor (SM) of a GH100 is depicted. One GH100 chip contains 144 of these SMs. The key features of the GH100 are summarized in table 2.3. The table only covers NVidia's FP32 units as they are essentially fused multiply-add (FMA) units performing two floating point operations (Flop) and are the most comparable to the floating point units HiPReP is using. More on HiPReP's floating point units can be found in section 3.2.2.

The peak performance of 51.2 TFlops gives a ball park number for a scaled out accelerator aspiring to contend with H100s when using single-precision floating point numbers.

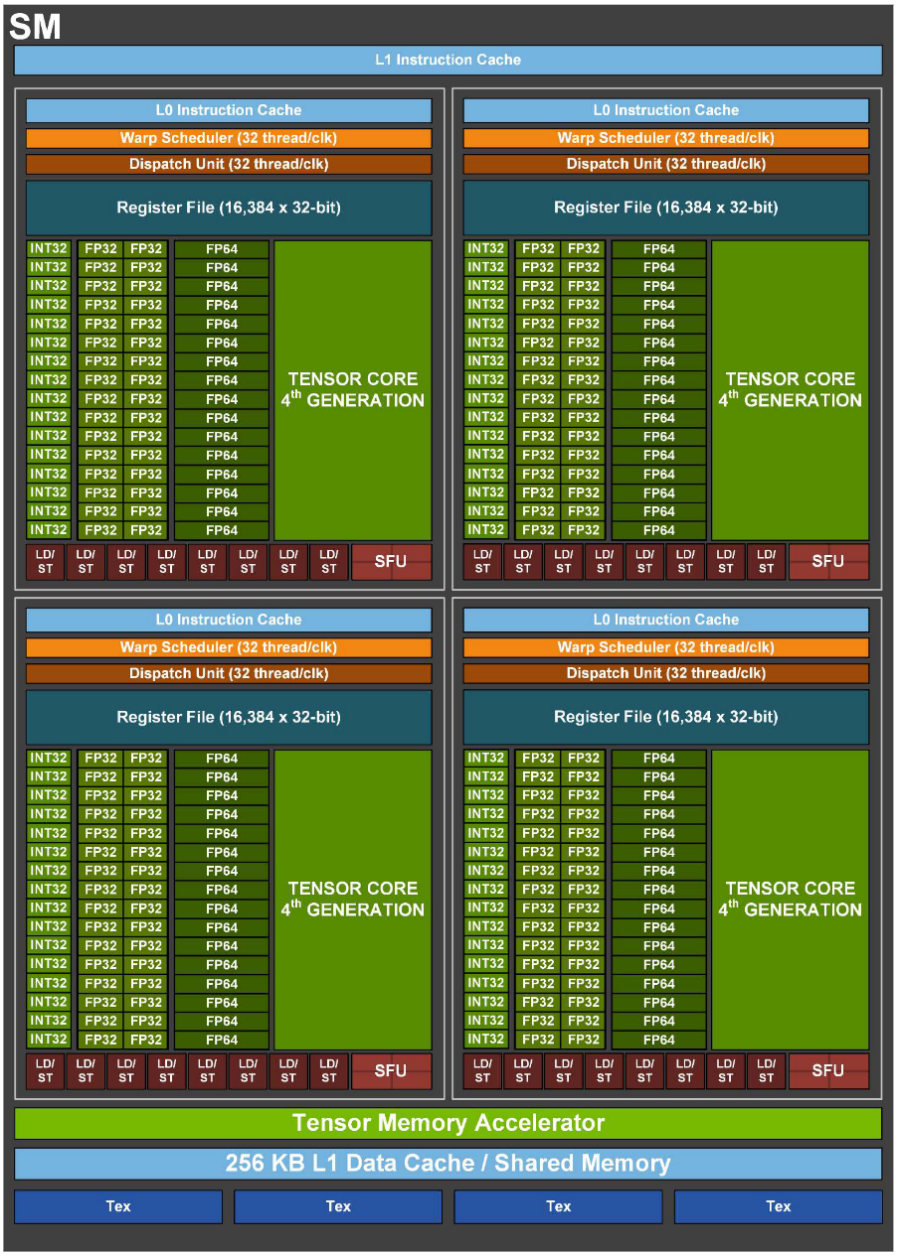


Figure 2.2.: "GH100 Streaming Multiprocessor (SM)" [32, Figure 7]

Further, all key features of the data sheet can be used to give an average utilization of each FP32 unit which is formalized in equation 2.1.

$$\text{utilization}_{\text{FP32}} = \frac{51.2\ \text{TFlops}}{144\ \text{SM} \cdot 4\ \frac{\text{TC}}{\text{SM}} \cdot 32\ \frac{\text{FP32}}{\text{TC}} \cdot 2\ \frac{\text{Flop}}{\text{FP32}} \cdot 1980\ \text{MHz}} \approx 0.886 \tag{2.1}$$

The utilization of the FP32 units can be used as an indicator for compute efficiency. This is important to know since it would be undesirable to reach the H100's performance with a multitude of FP32 units in another accelerator. Reaching the utilization of FP32 units in another accelerator is therefor a valuable point of comparison. In summary, a new HPC accelerator should achieve an average FP32 utilization of about $88.6\ \%$ and a total peak performance of roughly 51.2 TFlops to compete in the HPC domain as of today.

2.3. Optimization Flow

When optimizing a compute system, an application domain and an optimization goal is set. The application domain is represented by related benchmarks. An optimization goal can be computational performance. From there, an iterative process can be started to optimize the system, in software or hardware, and presuming an initial system is already working. The according flow diagram is shown in figure 2.3.

Starting from the initial design, the feature to optimize is evaluated with an appropriate evaluation methodology. Resulting from this, it can be determined whether the optimization goal is reached or not. If the goal is not reached, an appropriate improvement technique is applied and the design is evaluated again. Once the optimization goal is reached, the system is considered optimal for that defined goal.

Regardless of the methodology, the choice of applications is important to make informed decisions about which feature to optimize. When applying any methodology to one application run, the results are strictly *only* valid for a combination of compute system and software application. Generalizing from an analysis of one application can be deceiving, i.e. application A might run better on compute system 1 than application B, but the opposite might apply on compute system 2. For getting a broader view and understanding of a compute system, it is crucial to run many applications relevant to the end user. Otherwise, a compute system might be over-fitted to one application. Benchmark suites are used for further comparability between compute systems with respect to a specific application domain.

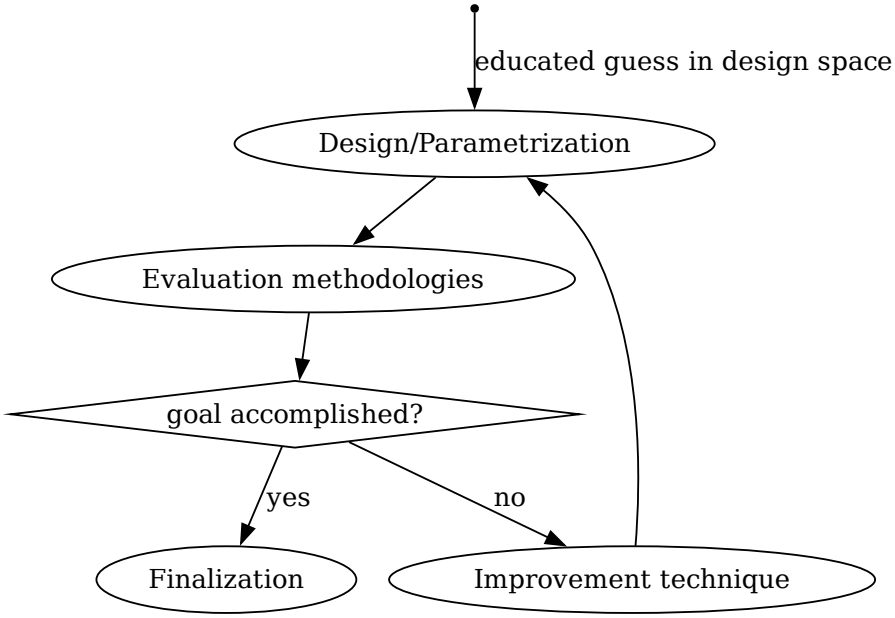


Figure 2.3.: Optimization flow diagram

In this section, the importance of evaluation methodologies for our approach is explained and which improvement techniques can be applied.

2.3.1. Evaluation methodologies

Evaluating the performance of HPC systems can be done in various ways. In general, strong scaling, weak scaling and the roofline model are used. Strong scaling is based on Amdahl's Law [33] and weak scaling on Gustafson's Law [34], while the roofline model is based on work of Williams' et al. [35]. In this work, these methodologies are used and they are explained in this section to understand the general idea behind them. We use the term *improvement technique* throughout this section as a general term for a change in the hardware system (or software) which improves performance. In section 2.3.2, we explain in more detail how these can be categorized.

These models are not competing concepts but rather complementing each other as they analyze different aspects of a compute system running an application. This is important to keep in mind because all models have their own purpose and also shortcomings - as George Box said: "All models are wrong, but some are useful" [36].

Amdahl's Law

Amdahl's Law evaluates the effective speedup which is attained when a certain improvement in hardware or software is applied on a fixed problem size. The increase in effective speedup is commonly referred to as *strong scaling*.

Most of the time, such improvements only affect a certain portion p of the total runtime, while the remaining portion of the runtime $s = (1 - p)$ is unaffected. In other words, the performance is first evaluated on a baseline and then the problem is solved again after applying the improvement to see how big the speedup is. The mathematical formula looks like this where the improvement is expressed as a factor n:

$$\text{speedup}_{\text{A}}(n) = \frac{1}{s + \frac{p}{n}} \tag{2.2}$$

As an example, a program was first evaluated on a single core system and an improvement parallelized a loop such that it runs on two cores now, making it twice as fast. The loop made up 60% of the runtime before. This yields:

$$\text{speedup}_{\text{A}}(2) = \frac{1}{\underbrace{(1 - 0.6)}_{=s} + \frac{0.6}{2}} \approx 1.42 \tag{2.3}$$

Even though we accelerated 60% of the runtime by a factor of 2, the overall speedup is 42%.

We can also derive a theoretical roof for the speedup from equation 2.2 which is defined by the portion of runtime unaffected by the improvement:

$$\lim_{n \to \infty} \text{speedup}_{\text{A}}(n) = \frac{1}{s} \tag{2.4}$$

This shows that any further scaling of the same improvement has less effect on the total runtime than before. For instance, applying more parallelism in the example before will only improve performance at higher costs. Instead, to further improve the runtime, it might be useful to reevaluate where most of the time is spent during execution and to improve the newly identified bottleneck.

Gustafson's Law

Gustafson observed that when the problem size *and* an improvement technique scale simultaneously, a quality slightly different from strong scaling can be assessed in compute systems which is called *weak scaling*. It concerns the question whether a problem scales linearly to which an improvement by factor n can be applied to a portion p of the runtime (where s is the unaffected portion of the runtime).

Rephrasing this, it is also desirable to sustain or reduce the total runtime when the problem size scales by scaling the according improvement. Again, the improvement will only affect a certain portion of the runtime p. The scaled speedup can be formulated with improvement factor n as follows:

$$\text{speedup}_{\text{G}}(p, n) = \frac{t(n)}{t(1)} = \frac{s + n \cdot p}{s + p} = s + n \cdot p \tag{2.5}$$

Concluding from this, when a problem size is scaled by p', ideally there exists an improvement with factor $n' \approx \frac{1}{p'}$ to make the system scale weakly and sustain the total runtime when increasing the problem size, even if it does not scale strongly. However, finding such a general improvement which always results in a linear speedup is impossible in practice. Rather, a combination of various improvements may speedup the runtime over a range of n relevant to the designer.

In HPC systems, even superlinear speedups have been reported and analyzed in [37], e.g. scaling the number of processors by n' to a scaled problem size, led to a speedup $> n'$ for certain ranges of $n_0 < n' < n_1$. The authors give recommendations on how to scale

- horizontally, i.e. increase the number of processors or accelerators, or
- vertically, i.e. increase cache capacity or levels in the memory hierarchy, or
- diagonally, which is a combination of the two other types,

based on the characteristics of an algorithm.

Empirically, scaling can be shown from measured runtimes by taking the measured runtime $t(1)$ of one work unit as baseline (i.e. the portion of the problem which is processed without the improvement), and dividing this runtime by the runtime $t(n)$ of the scaled problem. This yields the efficiency of each process:

$$\text{efficiency}(n) = \frac{t(1)}{t(n)} \tag{2.6}$$

When we achieve ideal weak scaling, $\forall n : \text{efficiency}(n) \approx 1$ applies.

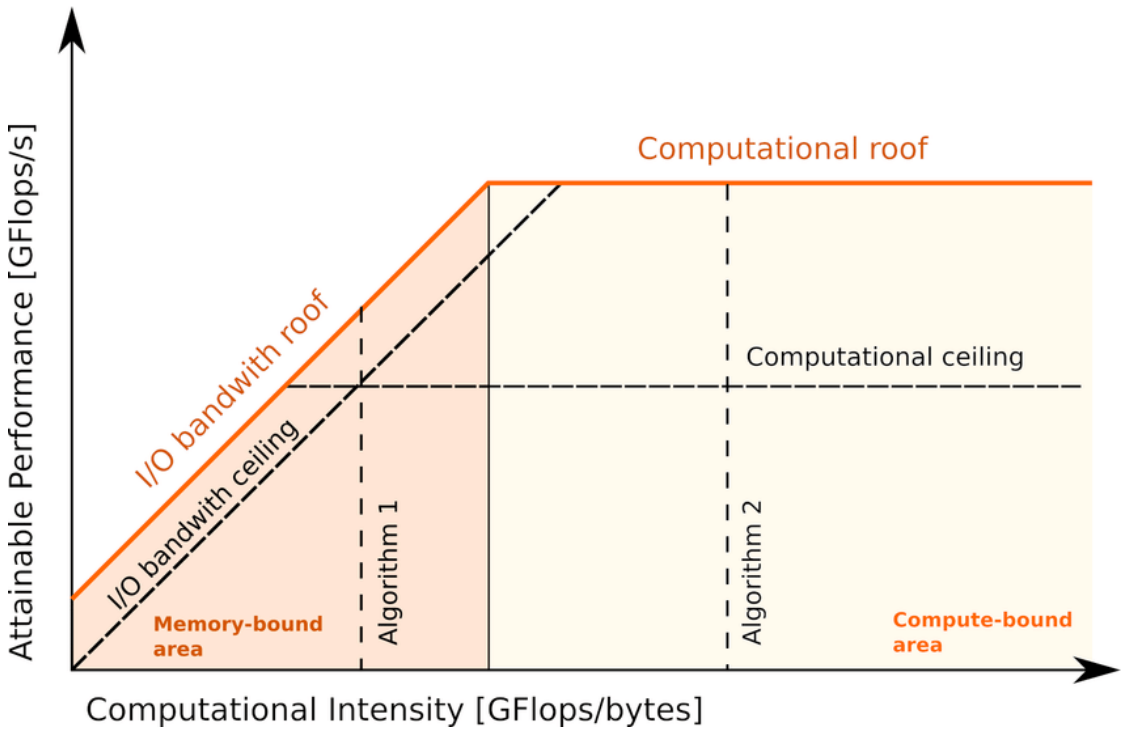


Figure 2.4.: Roofline model[3]

Roofline model for MMM

The roofline model was introduced by Williams et al. [35] and assists in identifying memory and compute bottlenecks. In figure 2.4, an explanatory roofline model is depicted: On the abscissa, the computational (sometimes arithmetic) intensity and, on the ordinate, the performance is plotted, both on logarithmic scale on the basis of two.

The computational intensity (in Flops/bytes) is an invariant property of an algorithm applied to specific data, aiming to capture their essence in a single number. In [38], for instance, the computational intensity is analyzed for (dense) matrix-matrix multiplication (MMM).

Given the bandwidths attainable in the system, e.g. memory or bus bandwidths, a slope can be drawn in the diagram. This line is called bandwidth roof. The theoretical peak performance of the attached processors or accelerators, is a horizontal line - called

computational roof - which continues from the intersection with the bandwidth roof. This intersection point is called *ridge point*.

The area below the bandwidth roof is referred to as memory-bound area, while the area below the computational roof is referred to as compute-bound area. Whenever an algorithm's computational intensity lies within one of these areas (s. Algorithm 1 and Algorithm 2), it shows at a glance how this application is bottlenecked by the system. Both roof lines are theoretical maxima. It's impossible to achieve higher performances than these roofs.

Depending on the application mapping, different performances can be reached (Computational ceiling). The goal is to get as close to the roofs as possible. When a point is farther away from the roofs, it indicates that improvements could be applied to bump up the performance number. Which improvement technique can be applied in a particular case is explained in section 2.3.2.

Looking at the kernels from a benchmark suite which represents the application domain to be targeted by the system, it is desirable to move the attained performance numbers as close to the ridge point as possible.

2.3.2. Improvement techniques for performance

Improvement techniques involve adding more *compute parallelism* to the architecture [39]. Compute parallelism can be exploited at different levels:

MLP memory-level parallelism: How many memory requests can be in flight at maximum?

ILP instruction-level parallelism: How many instructions can be handled by the processor pipeline at once, i.e. how deep is the processor pipeline? What is the issue width if it is superscalar? (Sometimes referred to as loop-level parallelism.)

DLP data-level parallelism: How many data words can be put in the processor pipeline at once at maximum? (Commonly exploited by vector extensions for ISAs of general purpose CPUs, e.g. AVX in x86 and V extension in RISC-V.)

[3] Creative Commons Attribution 4.0 International, accessed 25 January 2024. https://www.researchgate.net/figure/Roofline-model-based-on-49-The-x-axis-represents-the-operational-or-computational_fig2_362880728

TLP thread-level parallelism: (Special form of data level parallelism) How many threads operating in lock-step (i.e. executing the same instruction) can be issued at once? (Commonly exploited by GPUs. Not to be confused with simultaneous multithreading or hyper-threading, as seen in superscalar processors, which are actually techniques to improve ILP. However, SIMT architectures (single instruction, multiple threads) like GPUs combine TLP with with simultaneous multithreading/hyper-threading.)

Task task parallelism: How many tasks can be processed concurrently by the available processors or accelerators? (Task dependencies must be identified by compilers or programmers.)

According to Cabezas et al. [40], in the compute-bound area, this could mean that the level of ILP, DLP, or TLP needs to be increased. In the memory-bound area, MLP improvements like more levels of memory hierarchy, higher cache capacities or deeper buffers like of the miss status handling registers (MSHRs) should be considered. When the achieved performance cannot reach the system's memory bandwidth, task parallelism might be a good starting point.

2.4. Host system

In this section we describe the simulated host system in which our contribution is integrated.

Sections 2.4.1 and 2.4.2 contain excerpts from a paper submitted to the journal "Microprocessors and Microsystems" which is currently being reviewed. Quotation marks are omitted in these passages to improve the reading flow.

2.4.1. Chipyard

Chipyard [41] is a SoC generator framework written in Scala and Chisel [42]. It provides a variety of accelerators and RISC-V processors, as well as a memory hierarchy which can be configured, simulated, and synthesized. Chisel itself is a domain-specific language embedded in Scala.

One of the generators in Chipyard is Rocket Core [43]. Rocket Core is an in-order RISC-V core and provides the Rocket Custom Coprocessor (RoCC) interface [44]. It makes Rocket Core's L1 data cache (L1D), FPU, page table walker and tile link (bus between L1s and

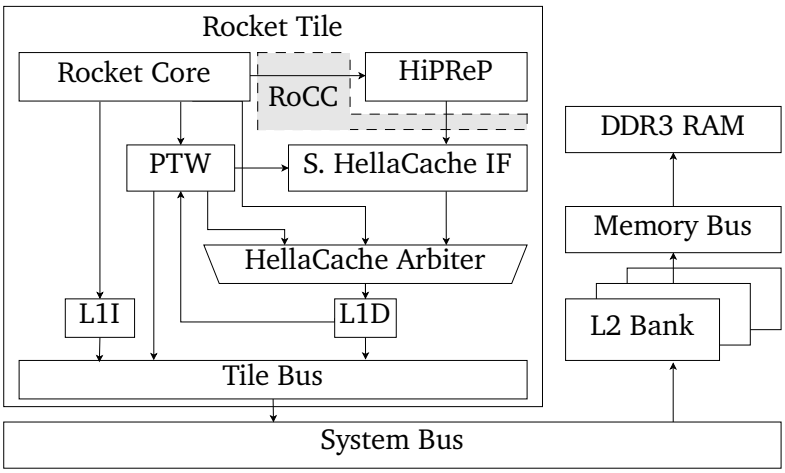


Figure 2.5.: Host system with one Rocket Tile

L2 cache) accessible. To add a new generator to Chipyard, it needs to be first put in the `generators` folder as a git submodule within the Chipyard git repository. By modifying `RocketConfig.scala` in `generators/chipyard`, for instance, the new generator can be used by adding a new configuration containing said generator. HiPReP's source code is available online at `https://github.com/PhilippKaesgen/hiprep`.

2.4.2. System and memory hierarchy

In figure 2.5, a block diagram of the system and memory hierarchy is shown in which HiPReP is integrated. HiPReP only requires access to the L1D through the SimpleHellaCacheIF module from Rocket and the Rocket's command interface. The block "RoCC" interfaces with the SimpleHellaCacheIF module from Rocket.

By accessing the memory like this, first, data is handled by Rocket's cache coherence protocol (Tile Link Protocol [45]), and, second, memory requests can utilize Rocket's Page Table Walker (PTW), to translate virtual addresses to physical addresses, and check access privileges. Hence, no additional hardware for the memory subsystem is required to integrate the HiPReP into Rocket other than specified by the block "HiPReP".

3. HiPReP

This chapter introduces the main components of HiPReP and explains how the HiPReP is configured by the host CPU, as well as how it loads, processes and stores data. In figure 3.1, an overview over the HiPReP architecture is depicted which expands the "HiPReP" block from figure 2.5. The dashed boxes refer to the components it connects to in the host system in Chipyard.

First, in section 3.1, an example application is mapped on a 3×3 HiPReP to sketch out the functionalities of the architecture and how the processing elements (PEs), address generating units (AGUs), frontend (FE), and SimpleHellaCacheIF interact with each other. In the remainder of this chapter, all components are described in detail.

The general design follows the producer-consumer pattern which means that the major components communicate with each other following the ready-valid handshake protocol. In appendix A, this protocol is explained in detail. All grey arrows implement the ready-valid protocol in the following block diagrams.

Note that sections 3.2, 3.2.2, 3.3, and 3.4 contain excerpts from a paper submitted to the journal "Microprocessors and Microsystems" which is currently being reviewed. Further, section 3.2.2 describes a mechanism which is published in [46], and a preliminary topology of HiPReP's CGRA has been presented in [47]. Quotation marks are omitted in these passages to improve the reading flow.

3.1. Motivating Example

Any application to be run on the HiPReP needs to be mapped first. A mapping defines where and when each computation of an application is done, i.e. on which PE in the CGRA, and which data are loaded and stored at which address in memory. An overview of

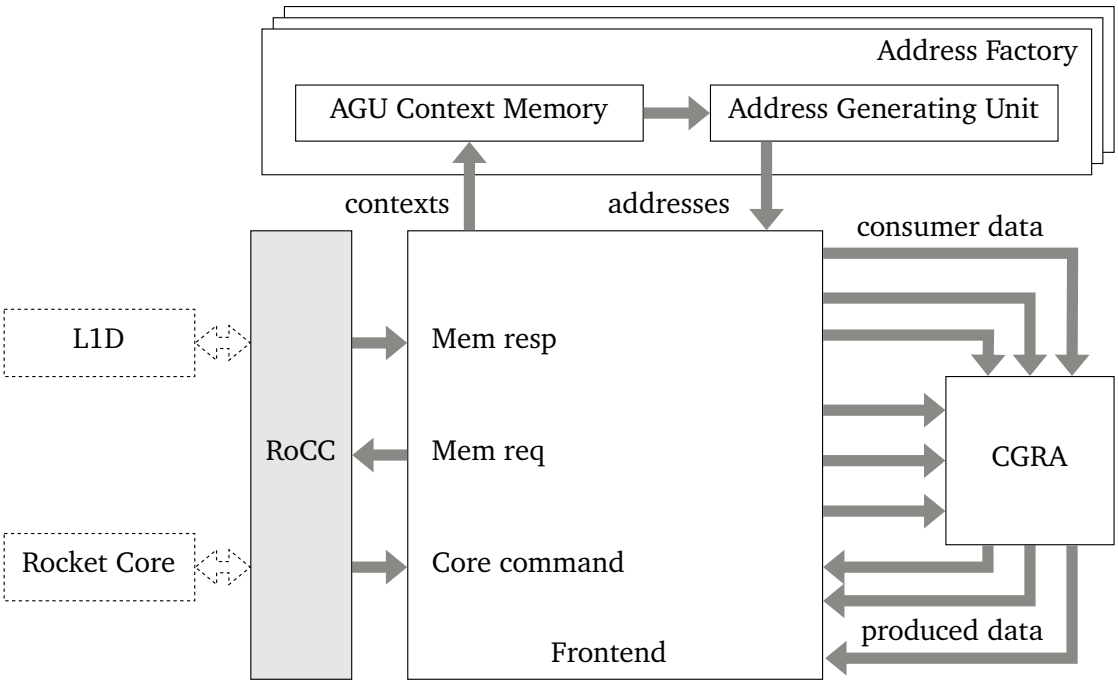


Figure 3.1.: Overview of the HiPReP architecture

the HiPReP architecture is given in figure 3.1. The dashed boxes refer to the components it connects to in the host system in Chipyard.

The mapping is usually done by a compiler but can also be done manually by a programmer. As an example, algorithm 5 is considered here. It is applied to the N elements of the three vectors a, b, and c yielding vector d.

Algorithm 5 Example

```
1: procedure EXAMPLE(N, a, b, c, d)
2:     for i in N do
3:         x = a[i] + b[i]
4:         y = x − c[i]
5:         d[i] = x ∗ y
```

A compiler takes a program, written in C, for instance, and performs several steps in the compiler frontend (lexical analysis, syntactical analysis, and contextual analysis) to

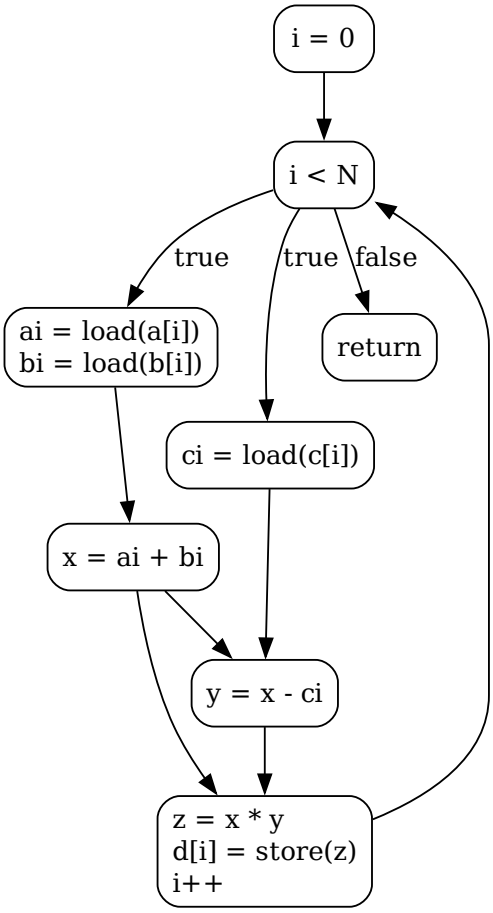


Figure 3.2.: Control data flow graph of the example application

finally generate a control data flow graph (CDFG). The CDFG of algorithm 5 is depicted in figure 3.2. CDFGs are directed graphs, where each node represents one or more instructions. The edges point to the next instruction in the execution sequence. On CDFGs, the compiler can perform code optimization techniques before it eventually emits machine code consisting of 0s and 1s, constrained by the available resources at a time. The machine code for CGRAs is called context.

To map such a CDFG, the available resources and their capabilities need to be taken into account, i.e. which PE executes an operation, and which AGU generates addresses to load and store values. In figure 3.3, the previous CDFG is annotated with a resource, assuming we have a 3×3 CGRA setup (other mappings are possible, too). The colors highlight the type of mapped resource: blue for AGUs in general, yellow as addition to read AGUs, red for write AGUS, and orange for PEs. The comma-separated numbers after "PE" indicate the PE's index on a two-dimensional grid. The instructions in the blue boxes are replicated in each AGU, i.e. each AGU increments the pointer to the next vector entry autonomously.

Following this, in figure 3.4, the actual mapping on the resources is shown: The AGUs generate addresses which are used for the memory requests. Loaded data is forwarded to the CGRA where they are processed. The results are then written back to the memory. Due to the ready-valid protocol, the PEs only process data when input data is available. Once the read AGUs do not load any more data and the last data word is written back to the memory, the AGUs trigger a response to the host CPU, concluding the task execution.

In the next sections, HiPReP's components are explained in more detail.

3.2. CGRA

In [47], a preliminary version of the HiPReP's topology was presented. In the final version, the topology of HiPReP is a mesh combined with long lines, only exposing CGRA outputs from the eastern PEs, contrary to the preliminary version where all PEs were able to write values to memory. This is a measure to reduce the wiring. In figure 3.5, a 3×3 CGRA is depicted.

3.2.1. Topology

The mesh connects each PE on a $r \times c$ grid with its eight nearest neighbors, shown as grey bidirectional arrows in figure 3.5. The unidirectional long lines connect the CGRA's

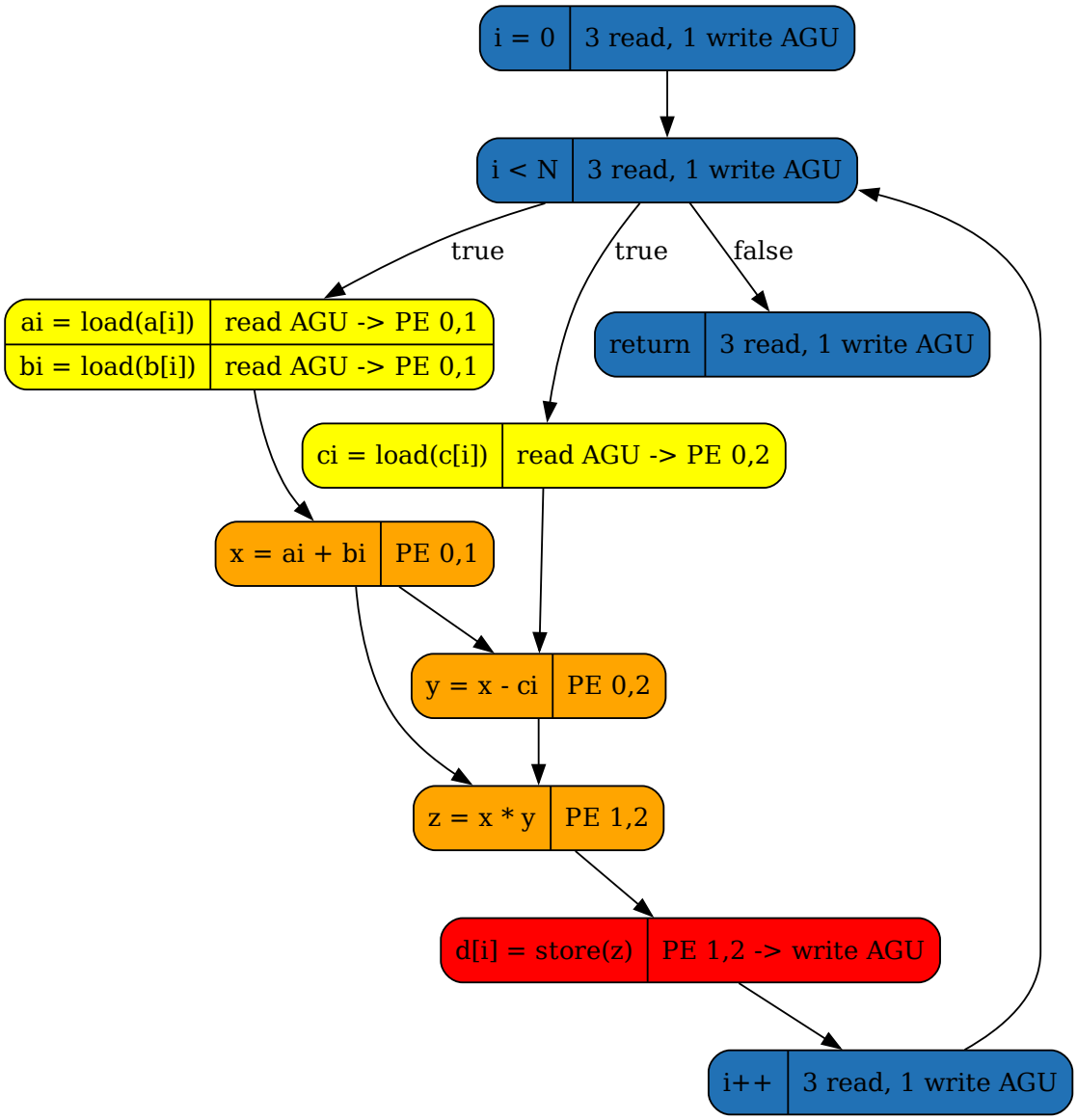


Figure 3.3.: Annotated control data flow graph of the example application

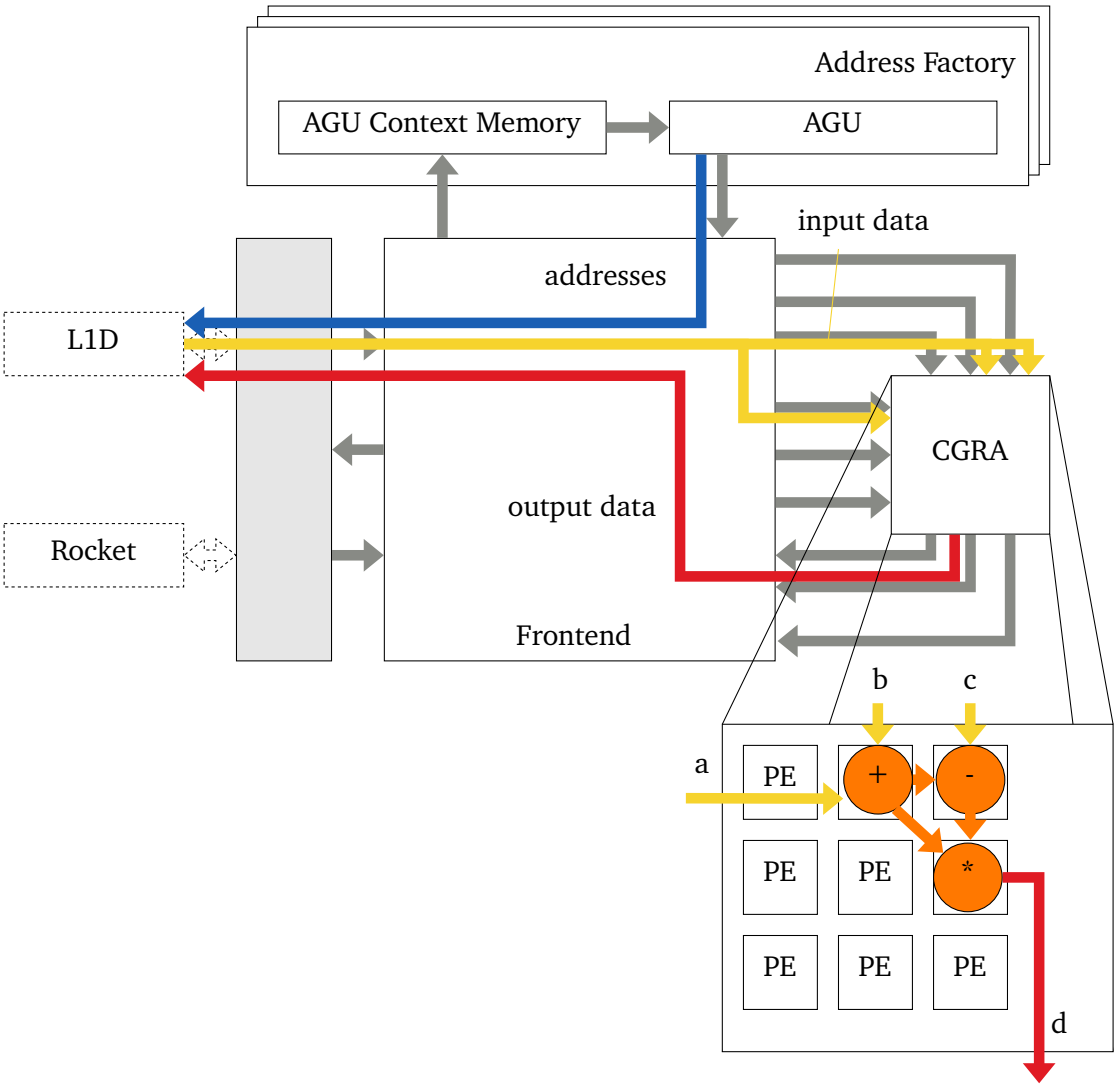


Figure 3.4.: Example mapping

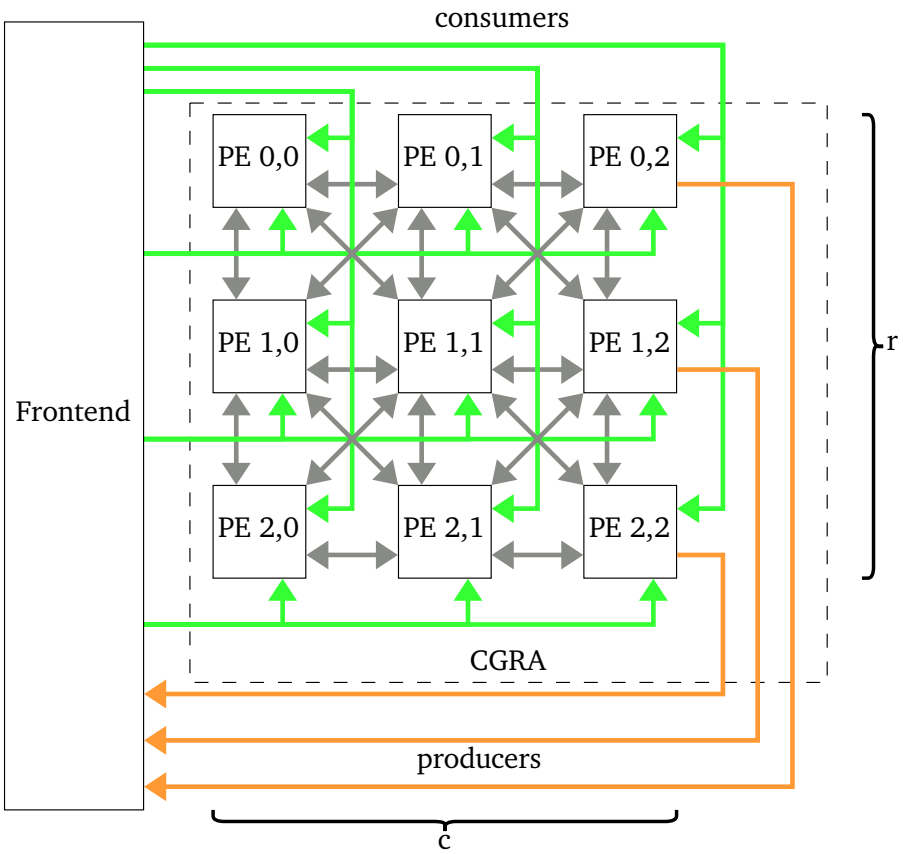


Figure 3.5.: HiPReP's CGRA topology

inputs with all PEs of a row or column, respectively, colored green. Via these inputs, the CGRA may consume values provided by the FE. The PEs of each row and column can be addressed over a bit mask, allowing broadcasts to all PEs or any subset in a row or column. The CGRA's outputs are highlighted in orange. Kernel mappings therefor need to forward produced values to these eastern PEs to send them to the FE to be written back to memory. A CGRA with $r \times c$ PEs has therefore $r + c$ inputs and long lines, and r outputs. PE 1,1 has ten inputs and eight outputs, while PE 0,2 has five inputs and four outputs, for instance.

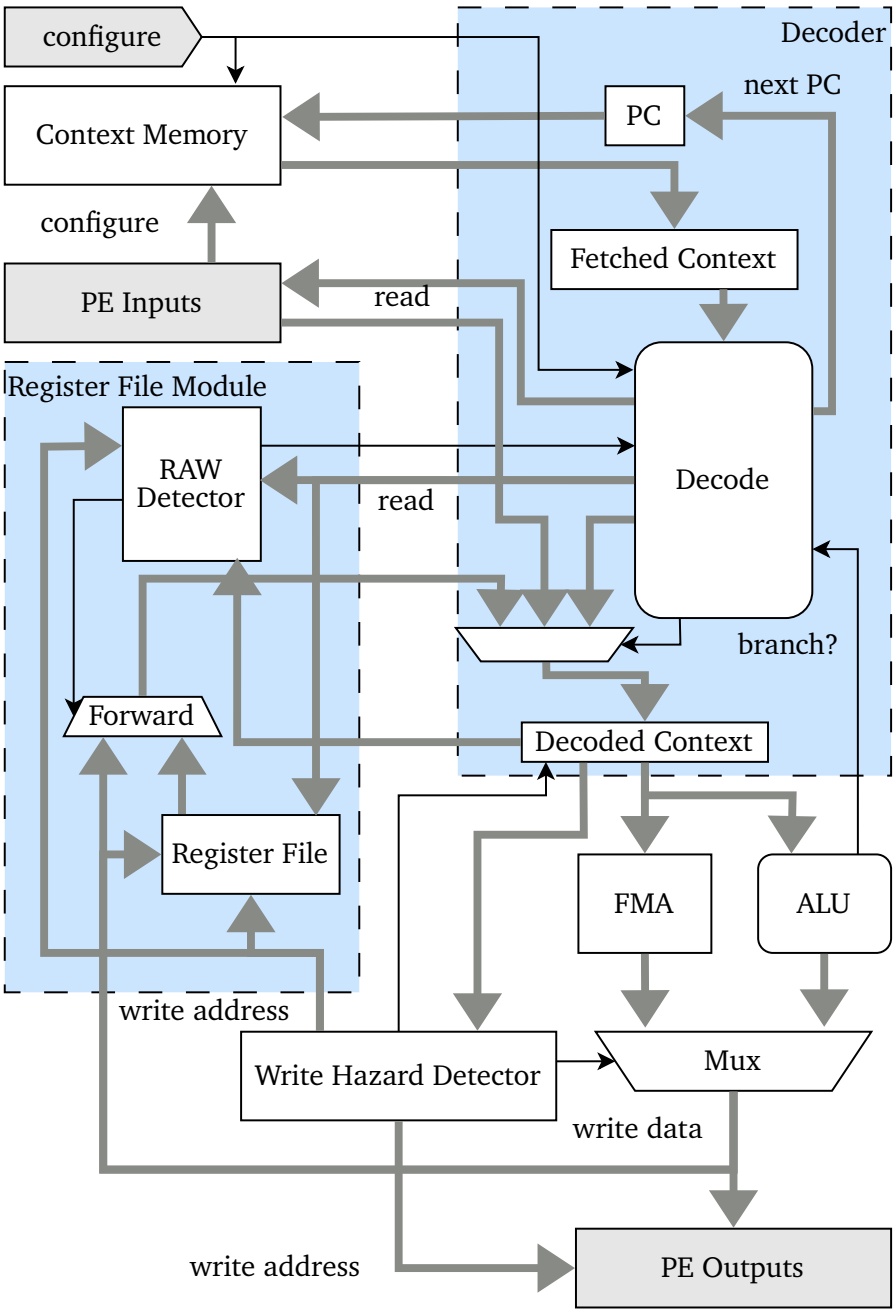


Figure 3.6.: Processing Element

3.2.2. Processing element (PE)

In figure 3.6, a PE is shown. HiPReP's PEs are templates which can be set up with different functional units. In this example, it contains an arithmetic logic unit (ALU) for integer values and a fused multiply add unit (FMA) [48] for floating point values.

PEs operate like in-order RISC processor pipelines with a context fetch, context decode, execute, and write-back stage. An exhaustive list of instructions is appended in appendix D. The PE inputs and outputs are special registers which can be read and written to, respectively. They can be addressed like the internal registers in the register file. Two inputs of each PE are connected to long lines, one to a row long line and one to a column long line. If the PE is located in the most eastern column of the CGRA, one output is connected to a CGRA's output. The FE described in section 3.4 handles these in a similar fashion as *stream semantic registers* introduced in [49]. The remaining inputs and outputs are connected to the neighboring PEs.

A PE context is like a RISC instruction consisting of an opcode, a target register address, and two source register addresses. It can perform arithmetic and logic operations on integers and floating point values. The source of the operand, i.e. input, internal register, or immediate value, are encoded in the PE contexts. Also it can handle control flow by processing conditional branches and jumps. In addition, a custom instruction is available, setting a maximum value for the program counter (PC), configuring the PE's PC to jump to a specified address whenever the former PC is reached without delay. This is a useful feature to deal with two nested loops within a PE. When the control unit switches to the "processing" state, the PE's and AGU's PCs are reset.

The PEs are capable of floating point arithmetic based on a FMA It is pipelined with ten stages for the UMC 65nm LL process node. Any floating point operation mappable to an FMA is performed on this unit, i.e. there is no dedicated floating point adder.

To handle long latency operations as performed by the FMA, the write hazard detector along with a read-after-write hazard detector are implemented [46]: Upon RAW detection, the decoder will stall the processing pipeline. The same strategy is applied to write-after write and structural hazards, unless the previous write can even be squashed by the new one.

Currently, in addition to the ALU and FMA, PEs can also be set up with an integer multiplier, as well as an integer-to-float and float-to-integer converter. Other types of FU can be implemented and added as needed.

3.3. Address generation

The AGUs used in HiPReP are implemented based on [50] and a block diagram is shown in figure 3.7. The main benefit is that with one context a two dimensional data structure can be traversed value-by-value. A read AGU generates a read descriptor consisting of a memory address and a bit mask. A write AGU generates a write descriptor which is a plain memory address. An AGU context consists of

base address first address on which further iterations will accumulate

N number of addresses to generate

stride the base stride to be added to the base address

skip to be used instead of stride when span equals zero

span indicates for how many times stride is used before skip is used once, to be repeated for as long as N is greater than zero

mask (only for read AGUs) endpoint descriptor, masking the addressed PEs of a row or column. The mask never changes for one context

When the counter reaches zero, the AGU requests the next context from the attached AGU context memory. The context from the memory is then written to the respective registers of the AGU. A base address equaling zero means that there are no further contexts to be processed. Then the AGU goes back into *idle* state. If the AGU belongs to a producer, this is forwarded to the FE's control unit. The control unit waits for all producers to finish before it goes back to *idle* state and sending a response to the host processor.

Together with an AGU context memory, an AGU builds up an address factory, as shown in figure 3.1. One AGU context can achieve different traversal paths through a memory space with the according set of parameters. Four example paths are illustrated in figure 3.8. The grid in the background represents a contiguous memory space, i.e. each tile represents a memory address. The address following the last tile of a row is the first tile in the next row.

The AGUs used in HiPReP are not tied to the general concept of the FE, so other approaches for the address generation could be dropped in their place as well, as long as one requester port always generates addresses on behalf of either one consumer or one producer. A consumer could therefor also generate an address based on a consumed value and send this address to a requester port. This would be a pattern as seen in indirect memory accesses.

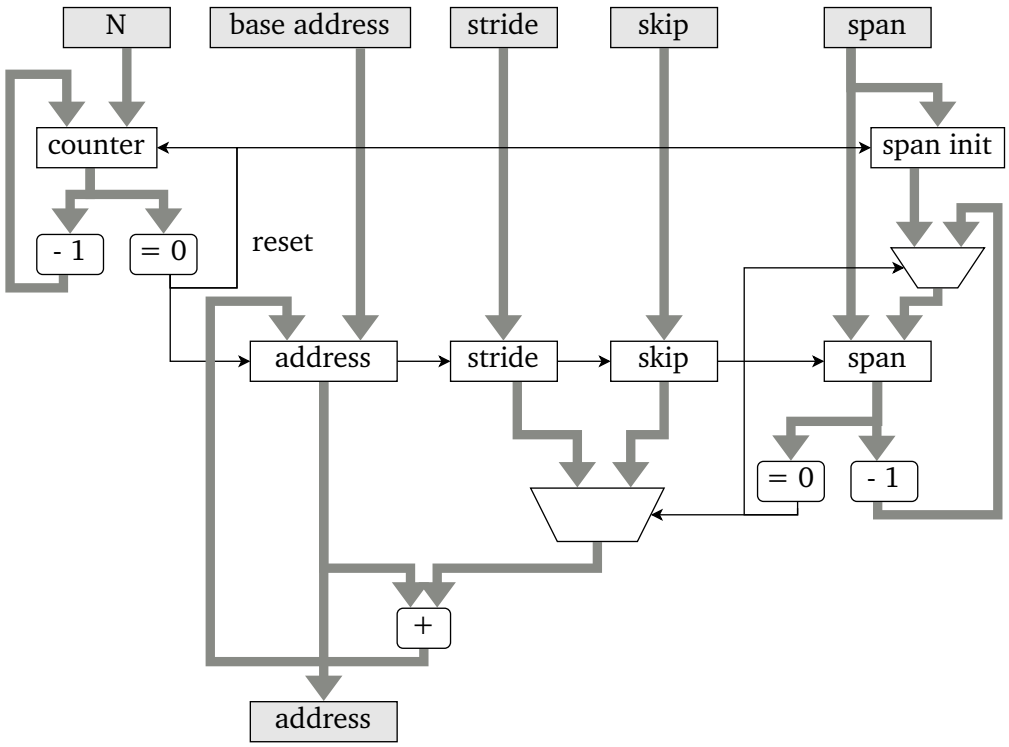
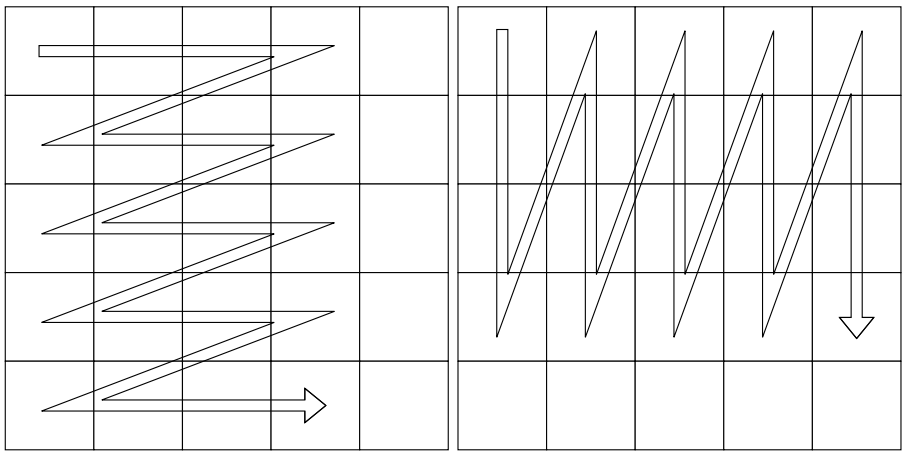
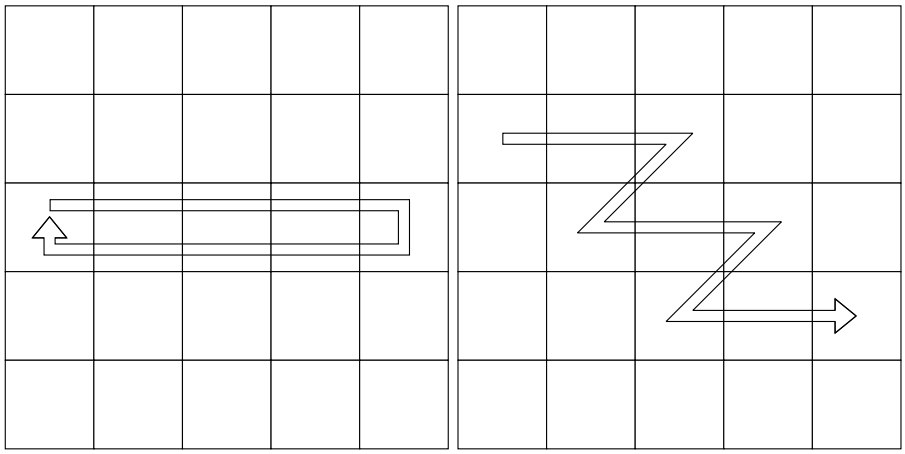


Figure 3.7.: Address generating unit (AGU)



(a) Row-wise
(N=20, stride=1, span=4, skip=2)

(b) Column-wise
(N=20, stride=5, span=4, skip=-14)



(c) Loop
(N=20, stride=1, span=5, skip=-5)

(d) Skewed
(N=9, stride=1, span=3, skip=4)

Figure 3.8.: Common access patterns enabled by the AGU

3.4. Frontend (FE)

The component bringing the HiPReP together with the host is the FE. It is the link between the HiPReP CGRA, AGUs, and the host system (figure 3.1). The FE orchestrates the memory accesses to the host and data supply to the CGRA. The FE exposes

- an interface compatible with RoCC,
- interfaces to consumers (CGRA inputs, shared by data and contexts),
- interfaces to producers (CGRA outputs),
- interfaces to the AGUs' context memories, and
- interfaces to the AGUs themselves (requesters).

The FE needs to handle multiple AGUs possibly sending addresses concurrently, enable loaded data arriving out-of-order to reach the CGRA inputs unhindered during the processing phase as describe in the motivating example in section 3.1. Also, before processing data the FE ensures that contexts are written to the respective context memories during the configuration phase. In this section, these mechanisms are explained in detail.

From a broad view, addresses are first generated by AGUs at the top of diagram in figure 3.1 and routed through the FE to the RoCC interface on the left for loading and storing data. Memory responses coming from the RoCC interface are then either concluded in the FE if they respond to write requests, or directed to the consumers at the right of the diagram if they respond to read requests. All producers and consumers (i.e. CGRA interfaces) have a dedicated AGU providing memory addresses for accessing the host system's memory. How exactly stores and loads are handled by the FE is explained in sections 3.4.1 and 3.4.2.

Internally, the FE needs to track the status of each memory access and to handle memory responses coming in different orders than their corresponding requests were sent, as explained in sections 3.4.1 and 3.4.2. Further, depending of the HiPReP's phase as described in section 3.4.3, it needs to handle the flow of data streams: From different sources, i.e. AGUs for memory addresses, PE outputs, and the RoCC for loaded data, to different targets, i.e. AGU memories, PE inputs, and the RoCC for data to store.

In this section, we walk through the FE with the help of the diagram in figure 3.9 which expands the diagram in figure 3.1 and show how write and read requests are handled, as well as how the HiPReP transitions between the three phases *idle*, *configuring*, and *processing*. We use store and write request synonymously, as well as load and read request, in the following. Also requesters are a more general term for AGUs, and are both used.

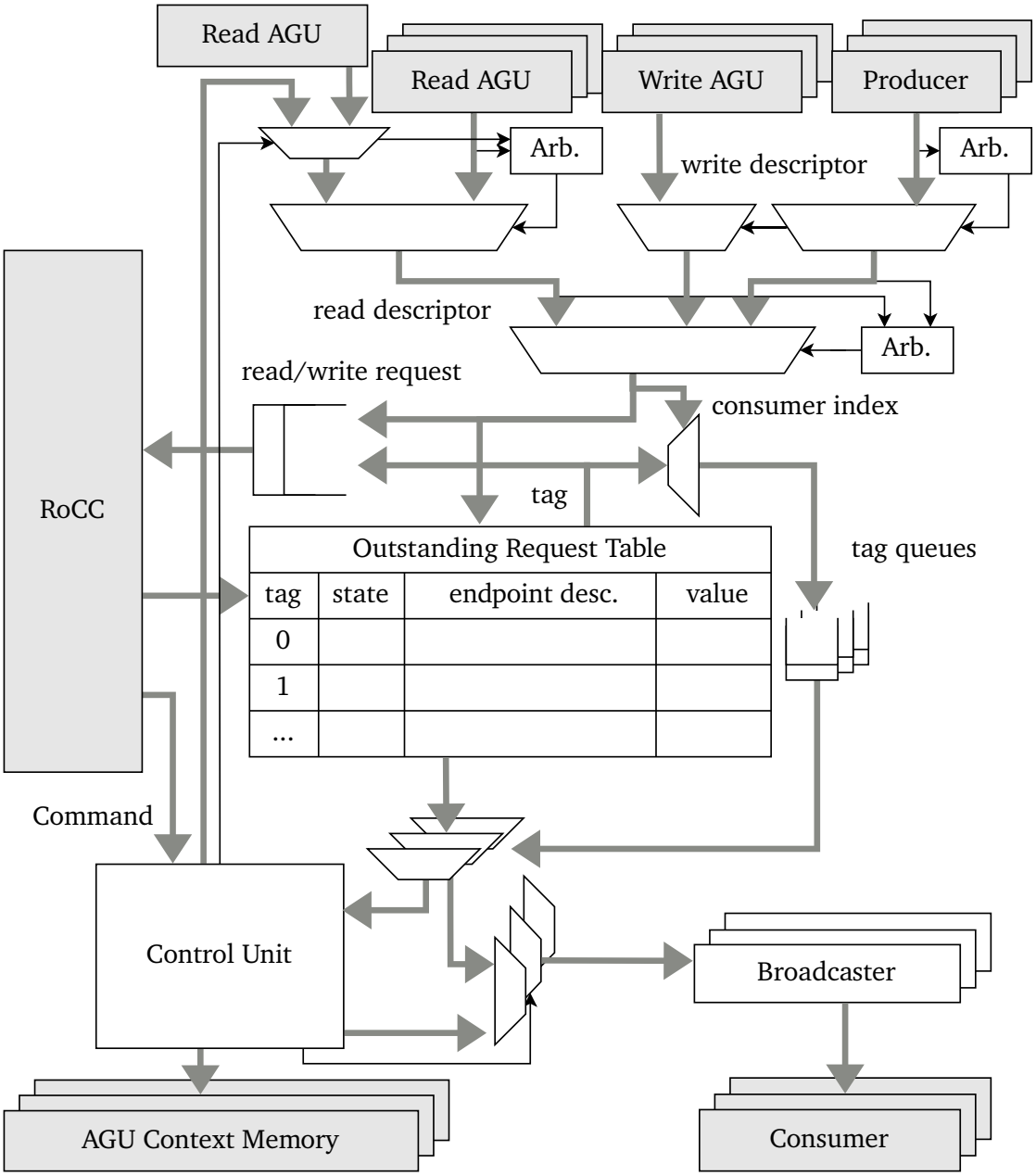


Figure 3.9.: The frontend

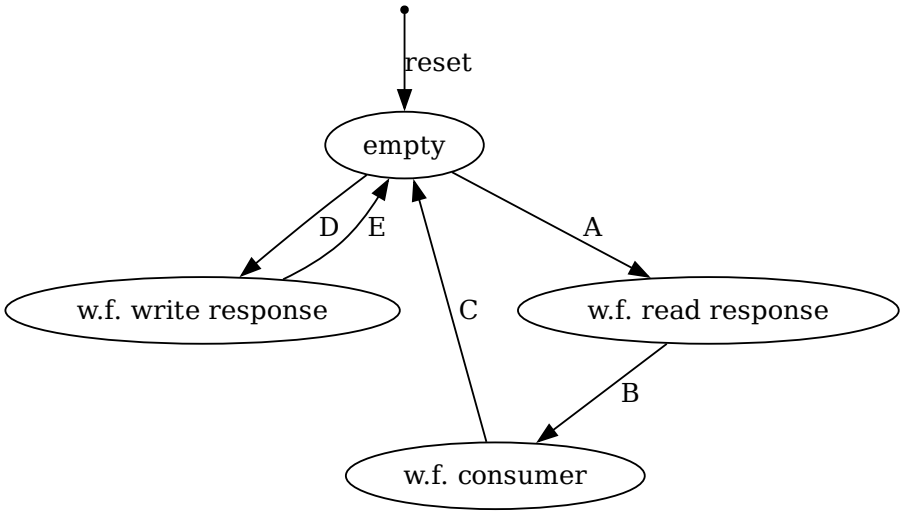


Figure 3.10.: State diagram of outstanding request table entries (w.f. = "waiting for")

3.4.1. Write requests

For writing, a producer conveys that it has a valid value. When this producer is arbitrated, the arbiter forwards the producer index to the multiplexer supplied by the write AGUs to select the associated write AGU. Both, address from the AGU and the producer value are merged and converted to a write descriptor. After granting access to the read/write request queue, the producer value and AGU value are being acknowledged at the sources and pushed to the read/write request queue with a tag. This tag is generated by the outstanding request table. It is an index pointing to an empty entry in the table. This entry's state transitions from *empty* to *waiting for write response* (edge D in figure 3.10) when any store request is pushed to the read/write queue. This allows to send as many requests to the memory as the RoCC interface allows. The RoCC interface will give back pressure to the FE if the L1D cannot handle more requests due to a full miss status handling register file. When the write to memory has been confirmed by a memory response (E), the response's tag points to the associated table entry, and the entry's state transitions from *waiting for write response* back to *empty* which completes the write to memory.

When multiple AGUs generate the same address, the FE does not detect collisions or a memory consistency model. It is up to the programmer or compiler to map kernels in a way that the addresses never overlap.

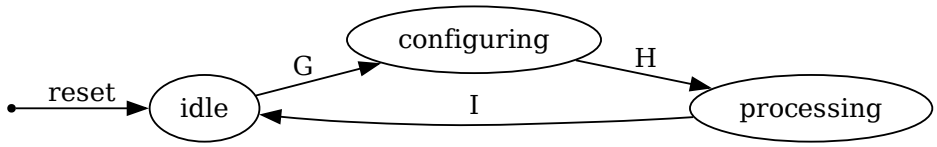


Figure 3.11.: State diagram of the FE's control unit

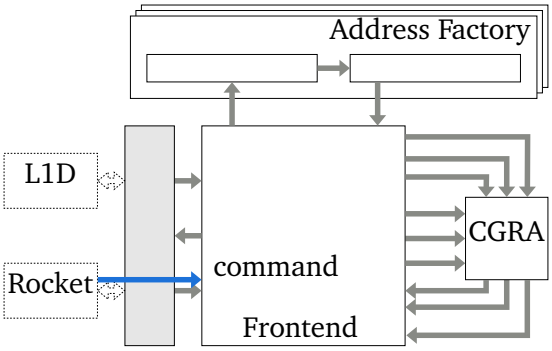
3.4.2. Read requests

For reading, a read descriptor is arbitrated which then follows the same path as the write descriptor until it reaches the read/write request queue, and the tagged table entry transitions from *empty* to *waiting for read response* (edge A in figure 3.10). At the same time, the request's tag is pushed to the tag queue which is associated with the targeted consumer. Thus, there are as many tag queues as there are consumers. The tag queues preserve the memory order for each consumer: When the FE receives the response to the read request, the corresponding table entry transitions from *waiting for read response* to *waiting for consumer* (edge B) while it puts the read data into the *value* field. In *waiting for consumer* state, the value is ready to be consumed: The tag queue's head selects the table entry and routes value and endpoint descriptor (i.e. the bit mask for indexing the PEs in a row or column of the CGRA) to a broadcast module.

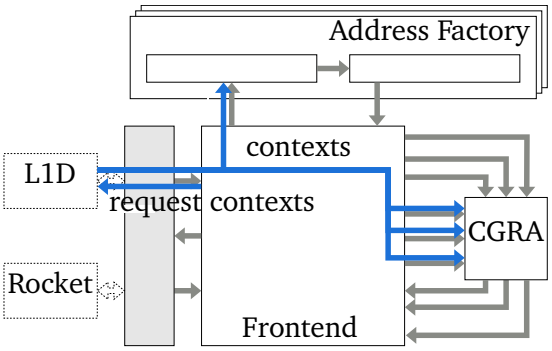
The broadcast modules are attached to the consumers, i.e. the CGRA inputs. Given a value and a bit mask, they ensure that every PE addressed by the bit mask reads the value and will give back pressure otherwise if any PE is not ready to consume the value. When all PEs consumed the data, the tag queue's head is dequeued and the table entry transitions from *waiting for consumer* to *empty*, completing the read from memory (edge C).

3.4.3. Control Unit

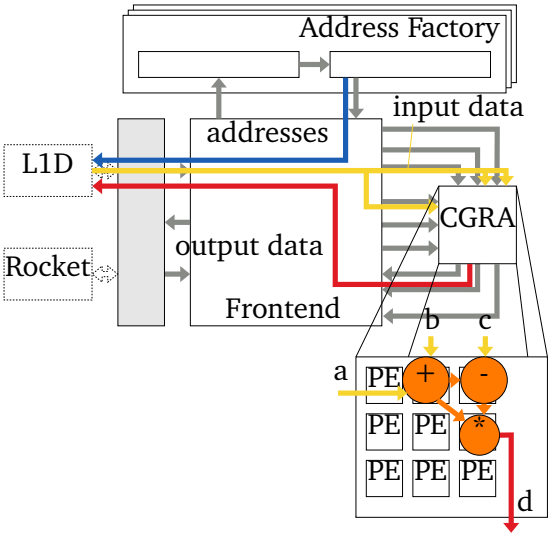
HiPReP's FE is controlled by a control unit. The state diagram is shown in figure 3.11 of which the edges are labeled (G), (H), and (I). When it receives a command from the RISC-V core through the RoCC, it sets HiPReP into the configuring state (edge G). As part of the this command from the core, the control unit gets a memory address pointing to the start of a HiPReP context, and the number of context words to fetch. In configuration state, the control unit will generate read requests and insert them like a regular read request at one of the read AGUs. The outstanding request table and the tag queue handle this read like a regular read request. When the table entry transitions to *waiting for*



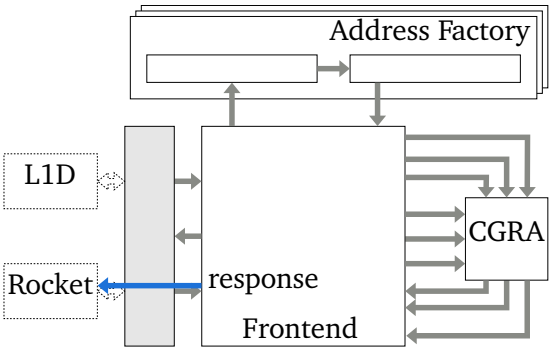
(a) RoCC command from Rocket core to HiPReP



(b) Configuration phase



(c) Processing phase $d = (a + b)\,(a + b - c)$



(d) RoCC response from HiPReP to Rocket core

Figure 3.12.: Example mapping

consumer the control unit intercepts the outstanding request table entry's output and distributes the contexts. The reason for handling contexts differently is that control words are embedded in HiPReP's configurations. These indicate to switch the configuration target, e.g. switch from AGU context memory 0 to AGU context memory 1. Configuration targets are always configured in the same sequence: First AGU context memories for the rows, then AGU context memories for the columns, context memories for the write AGUs, and then row-wise the context memories of the PEs (for an example configuration written in C, see algorithm 7). To configure the PE's context memories, the long lines are used which are wires connecting the FE with all the PEs of a row or column of the CGRA.

When the last context of a configuration has been written to the context memory the control unit transitions to the processing phase (edge H). In this phase, the AGUs start generating addresses as configured and the CGRA processes the data. The control unit monitors the write AGU inputs and write requests in flight. If there are no addresses left to generate and no write is in flight, the control unit considers the processing phase to be over, and sends a RoCC response while going back into idle state (edge I). Coming back to the motivating example from section 3.1, the entire flow would look like as in figure 3.12.

Often a user might want to reuse the previous configuration but execute it on other data than before. In this case, it is desirable to only patch the base addresses of the AGU context memories but leave the PE contexts as is. The control unit enables this by checking the number of remaining contexts to request from memory when it is about to change the configuration target from the last AGU context memory to the first PE context memory. If there are no contexts left to be requested or in flight, it will not reset the PE context memories and just transition from the configuring state to the processing state as described above. If there are still contexts to be requested or contexts in flight, it can recognize that these are for the PE context memories and that they are to be reconfigured. Consequently, it will reset the PE context memories before sending the new PE context words. (The reset logic for the context memories can be removed in the future, though, and the configurations need to conclude with an *END* instruction per PE instead, to prevent the PEs from execution and for reducing the PE area.)

3.4.4. Deadlock prevention

The FE implements a two-level arbitration process for choosing requesters: On the first level, it always prioritizes stores over loads. On the second level, it chooses requesters by round-robin among either producers or requesters for loads.

Writing values to memory works as follows: Whenever a producer is ready, the corresponding write descriptor is arbitrated among all write descriptors which might be valid and is granted memory access. This requires that the loads always leave at least one entry empty for stores in the outstanding request table.

When there is no write descriptor available, a read descriptor may be served. Since the AGUs can generate addresses ahead of the kernel running on the CGRA, the memory responses to loads might arrive before the CGRA might need them. Until they are consumed, the loaded values remain in the outstanding request table. This could lead the FE to deadlock when

1. the CGRA works on a compute-intensive kernel and cannot reserve an entry to write back at least one value because the outstanding request table is full with preloaded data, or
2. the consumers consume data at different paces. For example, two values are consumed from one consumer while only one is consumed from another consumer. This might cause imbalances when all requesters are arbitrated by round-robin.

Scenario 1. is impossible when $\#tags > \#consumers - 1 + len(tag\ queue)$, where

#tags is the number of entries in the outstanding request table and imposed by the tag width of the host system, and

tag queue is as long as required to cover the latency (in cycles) of a memory request on a L1D hit.

The formula accounts for round-robin access among all requesters, assuming all requesters always have a valid descriptor ready. This condition ensures that requesters for consumable values always leave one entry empty for a future store request.

Scenario 2. is countered by disabling a requester port when the head of the corresponding tag queue points to a value which is already waiting for HiPReP and the consumer port is not ready yet. This balances the arbitration of requesters.

In summary, the solution for 1. always leaves at least one entry for writing back a result from a long latency operation, and 2. balances the requesters as needed by the application.

Applied to the chosen host system, we have a L1D hit latency of six cycles, one cycle spent in each of the following stages:

1. the FE's request queue,

2. bypass of the translation look-aside buffer (TLB) lookup and access privilege check[1] (the TLB is part of the L1D in figure 2.5),
3. bypass of the page table walker (PTW)[2],
4. actual cache access,
5. FE's response queue, and
6. insertion of the response in the outstanding request table.

These stages are fully pipelined, making it possible to send a memory request every clock cycle. In the experiments conducted in this work, the HiPReP exclusively sends physical addresses but, due to Rocket's L1D design, memory accesses still go through the above pipeline, bypassing the address translation.

Rocket provides five bits for the tag, which caps the maximal number of entries in the outstanding request table at 32. Hence, we can have at most 25 consumers ($= \#rows + \#columns$, where the #rows is the number of rows in the CGRA, and #columns is the number of columns in the CGRA).

3.5. Deployment

In this section the deployment of HiPReP from the host processor - Rocket Core - is outlined. We detail how the HiPReP can be programmed in C and how the HiPReP configures itself, processes data accordingly, and notifies the host processor in Chipyard about task completion in section 3.5.1. In section 3.5.2, we explain how to deploy multiple instances of HiPReP in the same system. An instance of the HiPReP is always instantiated on a dedicated Rocket Tile (RoT). We refer to a single HiPReP as a single RoT setup, and to more HiPRePs to multiple RoT setup.

3.5.1. HiPReP configuration embedded in C

In Chipyard's `tests` directory, example source files in C are given which demonstrate how to deploy other accelerators. They provide convenience macros in different flavors, e.g. `ROCC_INSTRUCTION_DSS`, in `rocc.h` which are reused for HiPReP. The macros

[1] https://github.com/chipsalliance/rocket-chip/blob/v1.6/src/main/scala/rocket/NBDcache.scala#L745
[2] https://github.com/chipsalliance/rocket-chip/blob/v1.6/src/main/scala/rocket/NBDcache.scala#L758

are reads and writes to a custom control status register (CSR) which is connected to the RoCC interface. These macros enable calls to accelerators via the RoCC interface. An abridged C code is shown in algorithm 6.

The array `hiprep_contexts` contains the contexts to configure HiPReP and therefor how an application is mapped. As explained in subsection 3.4.3, first the AGUs are configured which are always at the beginning of the context array, followed by the PE contexts. Also, there are control words embedded for the control unit to switch the configuration target. In algorithm 7, an example of a 1×1 HiPReP context is shown, with comments for the keywords which cause the control unit to switch the configuration target, i.e. to the next AGU's or PE's context memory. Each PE can be programmed in a RISC-like assembly language wrapped in custom C macros. An exhaustive list of supported HiPReP instructions, i.e. context words, is included in appendix D. Since there is only one PE, there are only two read AGUs and one write AGU to be configured. For larger CGRAs, the contexts would contain more switch keywords.

The `ROCC_INSTRUCTION_DSS` macro then forwards the start address and length of the context array which the control unit uses to sequentially load the contexts. This macro will write a value to the register in which `failed` is located. Until the variable `failed` is used, the host processor may do other things. It is important to set a soft barrier in between consecutive to synchronize the hardware threads to guarantee that all RoTs have written back their results to memory. After the barrier, the RoT with hartid $= 0$ causes the main function to return while the others are trapped in a while loop.

When reusing the same configuration but for different data, the user or compiler can just update the base addresses in the `hiprep_contexts` in between runs and shorten `length` to the address of the last AGU context word. The PE contexts will persist in the context memories in this case. This reduces the reconfiguration time especially for large HiPReP setups.

In figure 3.13 a wavediagram of multiple HiPReP deployments in a row is shown with signals marking the command from rocket (edge A), configuration phase (edge B), idle (edge C), load available (edge D), and store available (edge E).

Whenever there is a command coming from the host, A is high and the HiPReP commences the configuration. B is high throughout the duration of the configuration. At the start of the simulation window, B is high for a significantly longer period than later. This is due to the PEs needing to be configured at the beginning. Later, the PE contexts are reused and only the AGU context memories are set up with new base addresses. One of the

Algorithm 6 Minimal example of host code deploying HiPReP (single RoT)

```
#include <stdint.h>
#include <stdlib.h>
#include "rocc.h"
#include "hiprep_isa.h"

/* input and output data */

/*
  Main function to be executed by a Rocket Core with hartid = 0
  in Chipyard
*/
int main() {

  /* Contexts for HiPReP's AGUs and PEs */
  uint32_t hiprep_contexts[] = { /* ... */ };

  /* return value set to fail at the beginning;
  will be overwritten by HiPReP or by a golden comparison
  */
  int failed = 1;

  /* determine start address and number of configurations */
  uint64_t start  = (uint64_t) hiprep_contexts,
           length =
               sizeof(hiprep_contexts)/sizeof(hiprep_contexts[0]);

  /* custom command from Chipyard's rocc.h in the tests directory */
  ROCC_INSTRUCTION_DSS(0, failed, start, length);

  return failed;
}
```

Algorithm 7 Example configuration for a 1×1 HiPReP

```
#include "hiprep_isa.h"  // HIPREP_INST macro

// ...
uint32_t  num_input_vals = 16,
          num_output_vals= 16;
uint32_t in[num_input_vals] = { /* ... */ };
uint32_t out[num_output_vals];

uint64_t start_address_input = (uint64_t) in;
uint64_t start_address_output = (uint64_t) out;

// ...

uint32_t hiprep[] = {                          // HiPReP configuration
    start_address_input, num_input_vals, 1, 0, 0, 1,
    0,                        // switch AGU; finish row AGU 0 contexts
    0,                        // switch AGU; finish col AGU 0 contexts
    start_address_output, num_output_vals, 1, 0, 0,
    0,                      // switch AGU; finish write AGU 0 contexts
    HIPREP_INST(MOVE, R1, I0, R0),            // start PE_0_0 contexts
    HIPREP_INST(MOVE, R2, I0, R0),
    HIPREP_INST(ADD_INT, R3, R1, R2),
    HIPREP_INST(MOVE, O0, R1, R0),
    HIPREP_INST(MOVE, O0, R2, R0),
    HIPREP_INST(MOVE, O0, R3, R0),
    HIPREP_INST(END, R0, R0, R0),
   -1                             // switch PE; finish PE_0_0 contexts
};
```

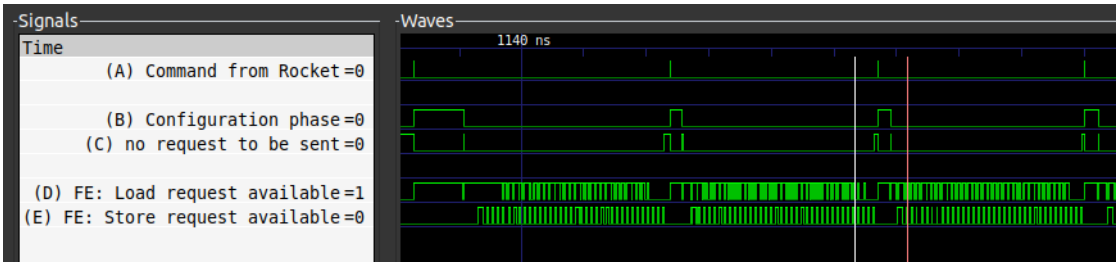


Figure 3.13.: Wavediagram of 3×3 HiPReP deployment

reconfiguration windows is framed by two markers. Signal C is high when the FE does not see any AGUs sending a valid address. This happens in two cases:

1. The HiPReP is idle or
2. briefly just after the configuration phase when contexts are still in flight internally.

The actual idle phase (1.) can be observed right before B is high. That is when the HiPReP called back the host and the host sets up the new base addresses. Case 2 occurs right after B is low again during the duration of one cycle.

During the configuration phase only loads are generated by the FE, and no data is written back. This can be seen at signals D and E. When the HiPReP is processing, both loads and stores are issued by the FE, with a few tailing stores toward the end of each processing phase, when all the input data is already loaded.

3.5.2. Multiple HiPReP instances in one system

As mentioned in section 2.3.2, one improvement technique for optimizing a computer system is task parallelism. Task parallelism requires that either the compiler or programmer splits an application in subtasks which can be executed concurrently on separate instances of a CPU or accelerator. In the Chipyard framework, we can generate multiple instances of RoTs with one Rocket Core and one HiPReP each. Then, we split the task in smaller tasks in software such that each instance of HiPReP produces a subset of the resulting data. On each RoT, the RoT-local Rocket core controls the RoT-local HiPReP. In the following, we use *multiple RoTs* and *multiple HiPReP instances* synonymously.

Algorithm 8 gives an abridged C code on how a program can be written to be executed on a system with multiple RoTs but omits the accelerator related code from algorithm 6. It is mostly adapted from Chipyard's `mt-hello.c` file. We will also omit the handling of task dependencies.

For running an application on multiple RoTs, first, a barrier is required (line 4) to synchronize all Rocket Cores on each RoT. Second, a `__main()` function (mind the prefixed underscores) needs to be implemented which is run on all RoTs with non-zero `hartid`. The `hartid` is a CSR which contains a unique identification number of each RoT, and is short for hardware thread identification number. The linker script instructs all of these non-zero hartid Rocket Cores to enter at `__main()`. The function name `__main()` is specific to Chipyard's linker script, so it could be different on other platforms. The Rocket Core with hartid 0 always enters `main()`. Common global preparations can therefor still

be executed on RoT 0 while the concurrent portion of the code is run on all RoTs. Based on the known hartid, a task can be partitioned in subtasks which then can be distributed, e.g. by splitting data sets into smaller pieces such that each core has a smaller workload in the end. To finalize a task, all cores synchronize on the barrier (line 30) and are kept alive (line 33) such that the core with hartid 0 can terminate the program. The actual HiPReP call can be placed in `__main()`, e.g. line 27.

Algorithm 8 Minimal example of concurrent execution in Chipyard

```
/* includes etc. */

/* barrier implementation from Chipyard's mt-hello.c */
static void __attribute__((noinline)) barrier() {
  static volatile int sense;
  static volatile int count;
  static __thread int threadsense;

  __sync_synchronize();

  threadsense = !threadsense;
  if (__sync_fetch_and_add(&count, 1) == n_cores-1) {
    count = 0;
    sense = threadsense;
  }
  else while(sense != threadsense);

  __sync_synchronize();
}

/* Main function directly called from hartids > 0 in Chipyard */
void __main() {
  /* Prologue: */
  int hartid;
  asm volatile ("csrr %0, mhartid" : "=r" (hartid));  // read hartid

  /* (code running concurrently on each hartid) */

  /* Epilogue: */
  barrier();  // let all harts (and foremost hart 0) sync

  if (hartid > 0)  // keep all harts alive such that only hart 0
    while (1);     // can terminate the program
}

/* Main function for hartid = 0 in Chipyard */
int main() {
  __main(); // also execute the same kernel as the other hartids
  return 0;
}
```

4. Evaluation

The HiPReP is evaluated by running the (dense) matrix-matrix multiplication (MMM) in two flavors: manually mapped, and compiler mapped. The reason for this approach is that the compiler, called CHiPReP [51], [52], is still a work in progress and it cannot exploit all features of the HiPReP architecture. For instance, CHiPReP cannot map two nested loops yet, or split kernels in multiple tasks such that the kernel could exploit task parallelism on multiple Rocket Tiles (RoTs). To also explore these features, a manually mapped MMM is used. In other words, this is at the present time the only way to explore the full performance on a single RoT (nested loop mapping) and multiple RoTs (manual task splitting). In table 4.1, the mapping features are summarized.

For completeness, a subset of CHiPReP compiled applications from Polybench [53] are included in appendix B, as well as finite impulse response filters and convolution filters in appendix C.

This chapter is structured as follows: First, the synthesis results for the UMC 65LL process are shown in section 4.1. Second, in section 4.2, the manual mapping and the CHiPReP mapping are detailed. Third, the host system parameters which are used in the evaluation are presented in section 4.3.

Sections 4.1 and 4.3 contain excerpts from a paper submitted to the journal "Microprocessors and Microsystems" which is currently being reviewed. Quotation marks are omitted in these passages to improve the reading flow.

Table 4.1.: Explorable features by mapping

mapping	single RoT	multiple RoT	kernel(s)	# mapped loop nests
manual	✓	✓	MMM	2
compiler	✓		MMM, FIR, 2D conv. filters, Polybench subset	1

Table 4.2.: CGRA Characteristics of HiPReP setups

Metric	1×1	2×2	3×3	4×4	5×5	6×6	9×9
Operand width (bit)	32						
Frequency (MHz)	793	793	793	752	704	666	555
Power (mW)	44	138	301	494	690	891	1690
Technology	UMC 65 LL						
Area (mm^2)	.257	.83	1.8	3.1	4.6	6.4	14.3

4.1. Synthesis results

For the design space exploration (DSE), we synthesized the 1×1, 2×2, 3×3, 4×4, 5×5, 6×6, and 9×9 setups, such that we can later attain comparable performance numbers and compare wall clock time rather than number of cycles we get from simulation. The runtime t of an application running on an $r \times c$ HiPReP setup is computed by $t = \frac{\#\text{cycles}}{f_{r \times c}}$, where $f_{r \times c}$ is the setup's clock frequency from synthesis. In table 4.2, the synthesis results of Synopsis Design Compiler are shown. The table is similar to the tables from Podobas et al. [54].

The results show that for 1×1, 2×2, and 3×3 the same clock frequency can be reached. This is because to that point, the FMA unit is on the critical path. Beyond 3×3, the long lines of the CGRA diminish the clock frequency.

4.2. Mapping of dense matrix-matrix mulitplication (MMM)

4.2.1. Manual

The manual mapping of MMMs is based on the algorithm 9. Since the loops are perfectly nested, we can fully utilize all features of AGUs and PEs. In the following, the mappings on one RoT and multiple RoTs are explained. Each RoT features its own HiPReP with the same setup. The local host CPU then configures its local HiPReP. For disambiguating RoTs and matrix tiles, RoTs are only referred to as such while matrix tiles are also referred to as simply tiles.

Algorithm 9 Naive MMM

```
1: procedure MATRIX_MULTIPLY(A_{M,N}, B_{N,K}, C_{M,K})
2:     for m in M do
3:         for k in K do
4:             C[m][k] ← 0
5:             for n in N do
6:                 C[m][k] ← C[m][k] + A[m][n] · B[n][k]
```

Single Rocket Tile

Mapping an MMM on HiPReP is best explained with an example but the principle can be applied to other matrix dimensions as well. For now we assume that the mapping always uses all PEs and AGUs, omitting corner cases where the matrices do not perfectly fit on the CGRA.

The mapping of an MMM $C_{16,16} = A_{16,4} \cdot B_{4,16}$ on a single RoT, i.e. one 4×4 HiPReP instance, is outlined in figure 4.1. It illustrates

- how the result matrix C is tiled based on the HiPReP setup,
- how one configuration traverses C,
- when the host CPU reconfigures HiPReP,
- how the AGUs traverse the input matrices A and B, and
- how the PEs consume and produce values.

First the result matrix C is divided into 4×4 tiles. Each of these tiles contains 4×4 values of C and can be computed at once by a 4×4 setup, e.g. $C_{0,0}, C_{0,1}, \ldots C_{3,3}$. Because we can handle two nested loops with one HiPReP configuration, we can configure HiPReP such that it can compute one row of C's tiles with one configuration. For computing the next row of C's tiles, the host only replaces the base addresses of the row AGUs which traverse A and the output AGUs which traverse C - the column AGUs' base addresses for B remain the same as we need to traverse B fully for each tile row of C.

In one HiPReP configuration, each row AGU loops as many times over one row of A as there are C tiles in one row of C. Each row AGU starts from the next row of A in an interleaved fashion. In each iteration, N addresses are generated by each AGU. Upon reconfiguration the traversal path stays the same, only the base address from where the

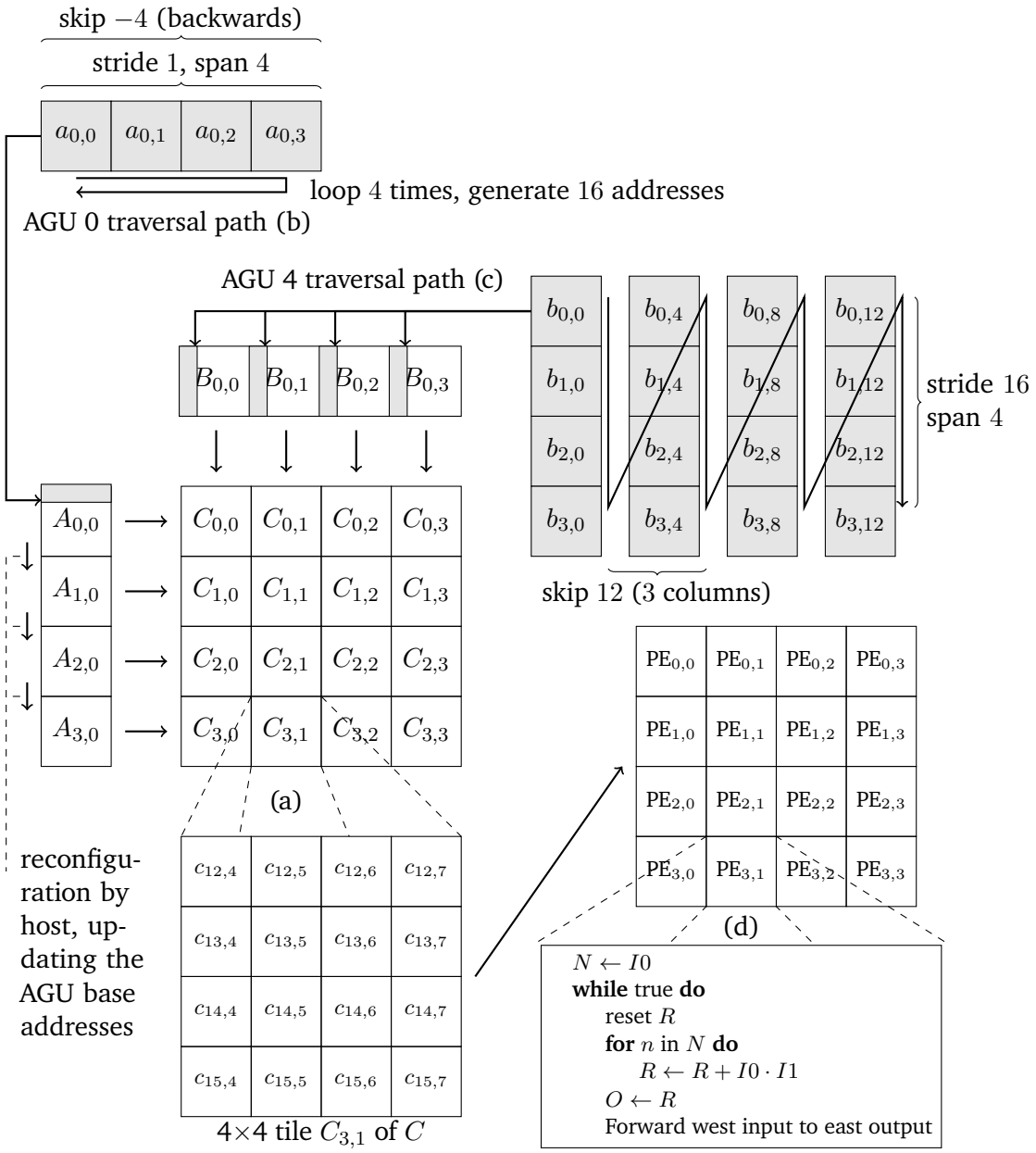


Figure 4.1.: Example of matrix tiling for single Rocket Tile, (a) 4×4 tiles directly mapped on 4×4 HiPReP; $C_{16,16} = A_{16,4} \cdot B_{4,16}$; (b) traversal paths of AGU 0 (row) and (c) AGU 4 (column) are shown, other AGUs use the same patterns starting from different base addresses; (d) one HiPReP configuration computes on row of C's tiles

traversal path starts is changed. In this case, the reconfiguration offsets 3 rows of A per AGU.

The traversal through B is done in a modified column-major order: Each column AGU strides to the matrix value in the row below and by reaching the last row, it skips to the first row but three columns to the right. Each column AGU starts from neighboring columns, i.e. consecutive values in the address space.

The PE configurations are oblivious of the tiling of C. They each accumulate N pairs coming from the long lines (I0 and I1), move the result to the neighboring PE in the East, and forward values from West to East. The most Eastern PEs push all values from one CGRA row to the FE to be written back to memory. Then, the PEs start anew to accumulate the next entry of C as long as the FE feeds them input data. In total, the CGRA produces $4 \times 4 \times 4$ output values with one configuration in this example.

Multiple Rocket Tiles

For multiple RoTs, basically the same mapping as for a single Rocket Tiles is used, except that we evenly split the result matrix C by tile row and compute each tile row on different RoTs: In figure 4.1, when we have two RoTs with HiPReP, the first two tile rows of $C_{16,16}$ are mapped on one RoT as one task, and the other two tile rows on the second as another task, effectively splitting C in an upper and lower half. When the setup has four RoTs, on each RoT one tile row is computed. The different base addresses for each tile are offset relative to the RoT's `hartid` in the C source file.

4.2.2. Compiled

The CHiPReP maps the MMM in a similar fashion on the HiPReP as the manual mapping does: It accumulates the products from A and B's entries in each PE before the CGRA moves all results to the most eastern PEs to be written back to memory. The only difference is that the AGUs only iterate over the inner most loop of algorithm 9 and, thus, CHiPReP's mapping requires to reconfigure HiPReP more often than the manual mapping. As a consequence, a comparison of these mappings mainly explores the impact of the AGU to handle two-dimensional and one-dimensional traversals through the memory space.

Table 4.3.: Simulated host system

Component	Specification
RAM	Micron DDR3, 2133 MHz 4 GiB, 4-channel, scheme7
L2	8 banks, 4 MB, 16 way
L1D	32 KiB, 64 sets, 8 way, block size = 64B
CPU	Rocket Core, (555..793) MHz

4.3. Host system parameters

For the experiments the HiPReP is integrated in a customized version of Chipyard (version 1.8.1 [41]) as shown in figure 2.5 and specified in table 4.3. The customization comprises the following:

- Modification of the SimpleHellaCacheIF of Rocket: The size of the replay queue is two by default. We ran an early experiment where we loaded the same cache line multiple times from HiPReP which should result in solely cache hits, apart from the initial miss. Since the HellaCache from Rocket might "nack" (not acknowledge) requests after two cycles, the replay queue in SimpleHellaCacheIF needs to wait one in three cycles to see if the request from two cycles before is "nacked" and must be replayed. This stall is propagated back to any RoCC accelerator and throttling the theoretically possible bandwidth by 33%. By increasing the replay queue's size to three, we can remove this bottleneck and harness the available MLP. To clarify, only RoCC accelerators are affected by this bottleneck - the host processors are not affected. This bottleneck is analyzed in detail in appendix F.1.
- DRAMSim2 configuration: By default, the emulated RAM only has one channel. We switched the DRAM configuration in the system.ini file to one which supports four channels to provide enough memory bandwidth.
- Addition of HiPReP as a Chipyard generator

The specs of the host system are very similar to Freedom U540[55], except that we chose a lower clock frequencies, depending on the setup. The corresponding frequencies of a tile are taken from table 4.2.

5. Results

In this chapter, the results of the experiments are presented. First, the results of the experiments on the single rocket tile are presented in section 5.1. Second, the results of the experiments on the multi-rocket tile are presented in section 5.2. Third, the HiPReP is compared with NVidia's GH100 architecture.

The experiments exploring the roofline model, strong scaling, and weak scaling can be reproduced using the scripts as provided by the project repository [1].

Sections 5.1.1 and 5.1.2 contain excerpts from a paper submitted to the journal "Microprocessors and Microsystems" which is currently being reviewed. Quotation marks are omitted in these passages to improve the reading flow.

5.1. Single Rocket Tile

5.1.1. Baseline comparison for MMM

First, we want to shed light on

- how well a few selected HiPReP setups compare against two baseline processors, and
- which HiPReP setups are selected for further exploration in the next section.

As baselines, we run the same MMM on Rocket Core [43] and *Berkeley Out-of-Order Machine (BOOM)* Core [56]. BOOM is an out-of-order RISC-V processor which also comes with Chipyard (to be precise: `LargeBOOMConfig`[2], a superscalar processor with

[1] https://github.com/PhilippKaesgen/hiprep

[2] https://github.com/riscv-boom/riscv-boom/blob/1606025fe0e0b10fcc45f4bd609a4fc30458054f/src/main/scala/v3/common/config-mixins.scala#L177

Table 5.1.: Feature matrix of runtime diagrams

data type	loop unrolling? no	yes
int	figure 5.1	figure 5.2
float	figure 5.3	figure 5.4

a pipeline handling three issues for integer operations at once, and single-issue for floating point operations).

For the following experiments, two matrices A and B are multiplied, like $C_{18,18} = A_{18,N} \cdot B_{N,18}$, and sweep over N to increase the computational intensity. For each N, N multiplications and $N - 1$ additions are required to compute an entry in the result matrix C. The matrices are stored in row major format.

A naive implementation as described in algorithm 9 is executed on Rocket and BOOM, i.e. without library support to optimize it. This assures that on the HiPRePs and the baselines the same number of operations is performed. Yet, the baseline is evaluated without and with loop unrolling. The version without loop unrolling is general, though, because it does not optimize for these exact matrix dimensions.

For the HiPReP, the kernel execution works as follows: Each configuration of a rxc HiPReP can compute r rows of the result matrix C. In other words, the HiPReP executes the two inner loops of algorithm 9 and the outer loop is executed on the host processor which adapts the base addresses. The reconfiguration time is included in the runtimes and not omitted. Depending on the HiPReP setup, we need to reconfigure HiPReP $\frac{18}{r}$ times since C has 18 rows in these experiments. This also means that small HiPReP setups benefit from loop unrolling of the outer loop more than big HiPReP setups due to more outer loop iterations on the host processor.

For convenience, we give an overview of the features we explore in figures 5.1, 5.2, 5.3, and 5.4 in table 5.1. In each figure, we plotted a bar diagram for the runtime in number of cycles. Each group on the abscissa is run on the same value N, and within one group we can see the runtimes on HiPReP setups and baselines. A horizontal dotted line is drawn to indicate the runtime of the fastest baseline within a group. If a bar is below that line the deployment of the HiPReP setup pays off, and otherwise not.

In figure 5.1, the runtime on Rocket significantly increases when we scale up N. BOOM's runtime is not affected as much. For both baselines, the scaling can be tracked.

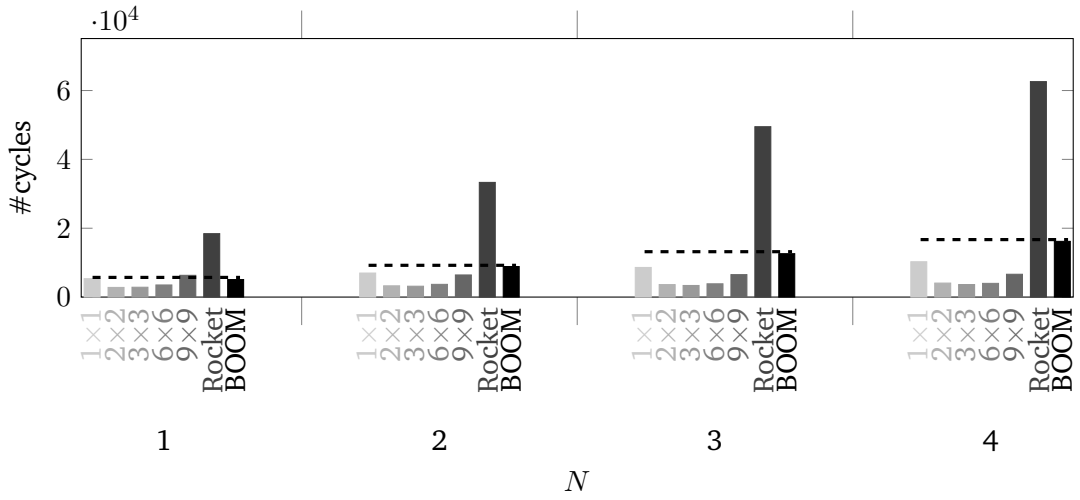


Figure 5.1.: Runtime comparison of MMM $C_{18,18} = A_{18,N} \cdot B_{N,18}$ (integer) including reconfiguration time for HiPRePs without loop unrolling

The 1×1 runtime increases more than the other HiPReP runtimes when we sweep over N. This is because it is subjected to more reconfigurations. 2×2 and 3×3 run even faster. 6×6 and 9×9 run longer than 3×3 for the chosen N.

The shorter runtime of 1×1 compared to Rocket can be explained by the parallel address generation and compute on HiPReP. On Rocket, address generation and actual compute are performed in one in-order pipeline; BOOM features an out-of-order pipeline, and address computation can overlap with actual computations. For the naive case without loop unrolling, the HiPRePs perform better than BOOM and Rocket.

When comparing figure 5.2 and figure 5.4, it can be observed that the baselines perform better when loop unrollings are turned on. Still, with increasing N, the baselines' runtimes increase more than the HiPReP setups' runtimes, showing that the FE and CGRA can service increased compute and memory demands better than the RISC-V processors.

In figure 5.3, we ran the same experiment as before, but with single precision floating point numbers. Despite floating point operations being multi-cycle operations, i.e. long latency, the overall runtimes behave similarly to for integer operations. For 1×1, the runtimes take notably longer compared to their integer counter parts due to the multi-cycle nature of FPUs. We do not see a huge difference when we compare the integer and floating point

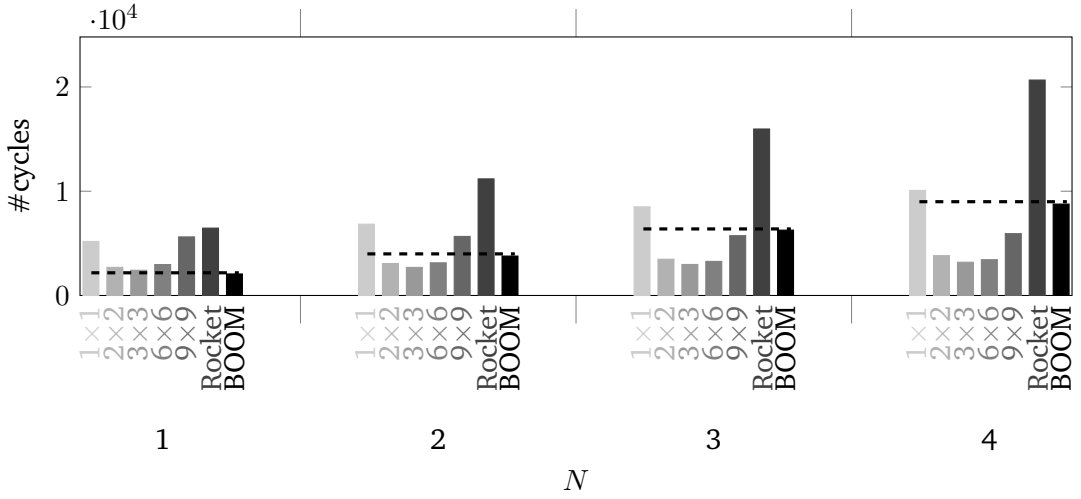


Figure 5.2.: Runtime comparison of MMM $C_{18,18} = A_{18,N} \cdot B_{N,18}$ (integer) including reconfiguration time for HiPRePs with loop unrolling

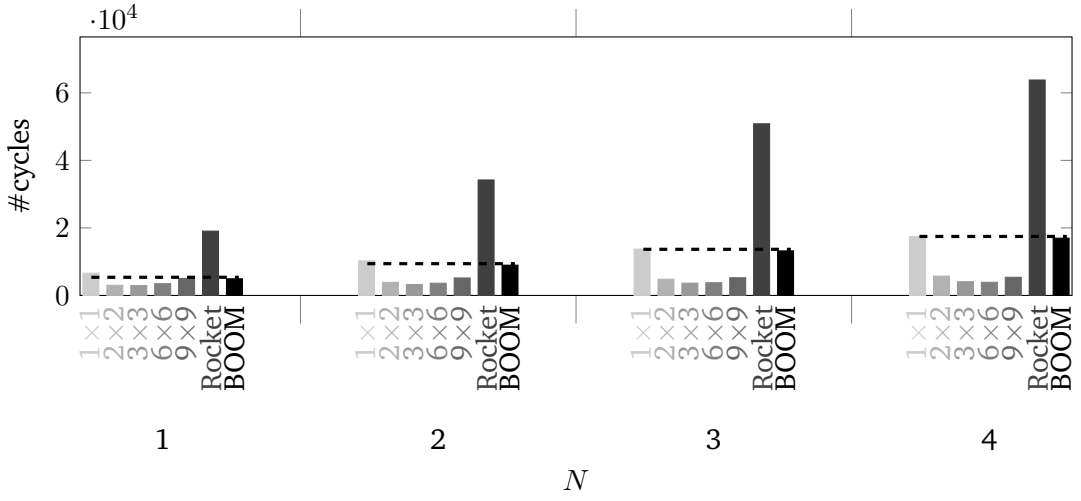


Figure 5.3.: Runtime comparison of MMM $C_{18,18} = A_{18,N} \cdot B_{N,18}$ (float) including reconfiguration time for HiPRePs without loop unrolling

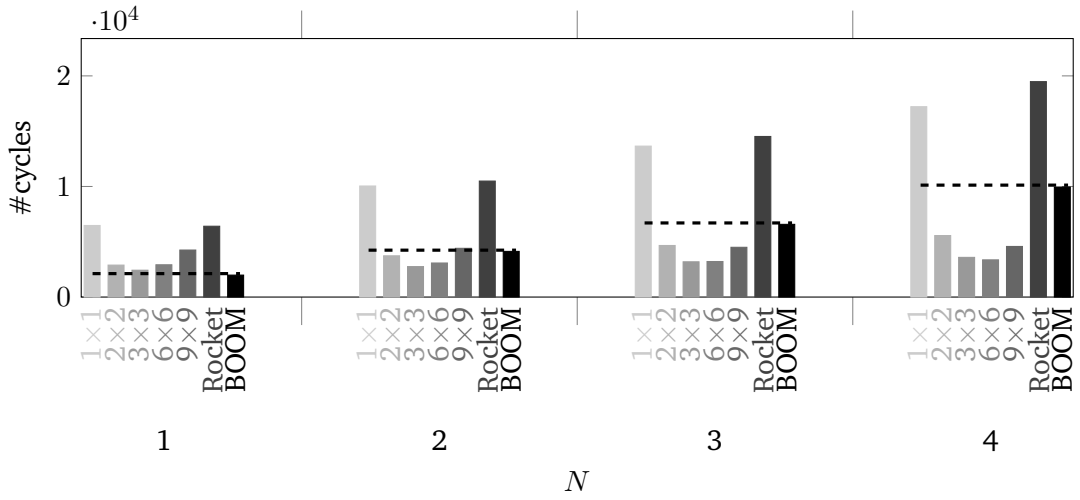


Figure 5.4.: Runtime comparison of MMM $C_{18,18} = A_{18,N} \cdot B_{N,18}$ (float) including reconfiguration time for HiPREPs with loop unrolling

runtimes when running on the other HiPReP setups. In figure 5.4, the results of the same experiment but with loop unrolling is shown.

Based on figure 5.2 and figure 5.4, it can be seen that despite of loop unrolling, the configuration overhead becomes less important with increasing N and the increased computing resources pay off. At $N = 2$, the 2×2 HiPReP setup's runtime, for instance, is already slightly better than the BOOM's runtime. For greater Ns, the HiPReP setups scale better than the baselines. So we can conclude that for the matrix-matrix multiplication, deploying the 2×2 HiPReP pays off compared to running it on boom when $N = 2$ which requires $(2 + 1) \cdot 18 \cdot 18 = 972$ operations (or more) and $2 \cdot 18 \cdot 18 = 648$ loads (or more). Below that number, it is better to run the application on a BOOM processor.

In conclusion from these charts, all HiPReP setups perform better in terms of #cycles for all analyzed MMMs where $N > 1$, and the 3×3 and 6×6 setup perform best among the preselected setups. In the following section, we will explore 3×3, 4×4, 5×5, and 6×6 to identify the setup which yields the best trade-off between setup size and throughput efficiency.

5.1.2. Throughput for MMM

To identify a HiPReP setup which can fully utilize the available memory bandwidth optimally, we run a bigger MMM ($C_{60,60} = A_{60,N} \cdot B_{N,60}$) than before to reduce the impact of reconfiguration. The considered setups are 3×3, 4×4, 5×5, and 6×6 now. In figure 5.5 the runtimes are plotted. Contrary to the previous plots, we plotted the runtimes in nanoseconds according to the synthesis results. Note that the HiPReP frequency decreases for setups larger than 3×3, hence the benefits of reduced cycle counts are getting smaller.

By looking at the runtimes only so far, we treated the HiPRePs, and hence the FEs, like blackboxes. Taking a closer look at finer statistics on how the FE interacts with the L1D, gives a better insight on the processing efficiency which is a result of the FE sending out as many requests as possible to the L1D.

We analyse the FE efficiency by breaking down the runtime at FE as follows:

idle No request is generated either due to no load/store available or the consumers giving back pressure.

back pressure The memory cannot accept further requests from the FE due to being busy with handling previous requests, while the FE has a valid request ready to be sent.

sending The FE can successfully send memory requests.

When we maximize the *sending* time, while reducing the *idle* and *back pressure* time across all setups, we deem the FE as running at the highest efficiency (for this application). For better comparability, we normalized these breakdown times to the respective total runtimes in figure 5.6.

In the breakdown, we can see that the 3×3 is *idle* for significantly longer periods than the other setups, while the *back pressure* time is the shortest. The *sending* time peaks for 4×4, closely followed by 5×5. For 6×6, the *sending* drops due to *back pressure*. Since the 4×4 has a slightly higher *sending* time than 5×5, 4×4 offers a better efficiency per PE for this MMM.

5.1.3. Comparison of manually mapped and compiled MMM computations

Although we stated previously that the CHiPReP is still a work in progress, and only features as describe in table 4.1 are supported, we compare the results for MMM $C_{8,8} = A_{8,48} \cdot B_{48,8}$

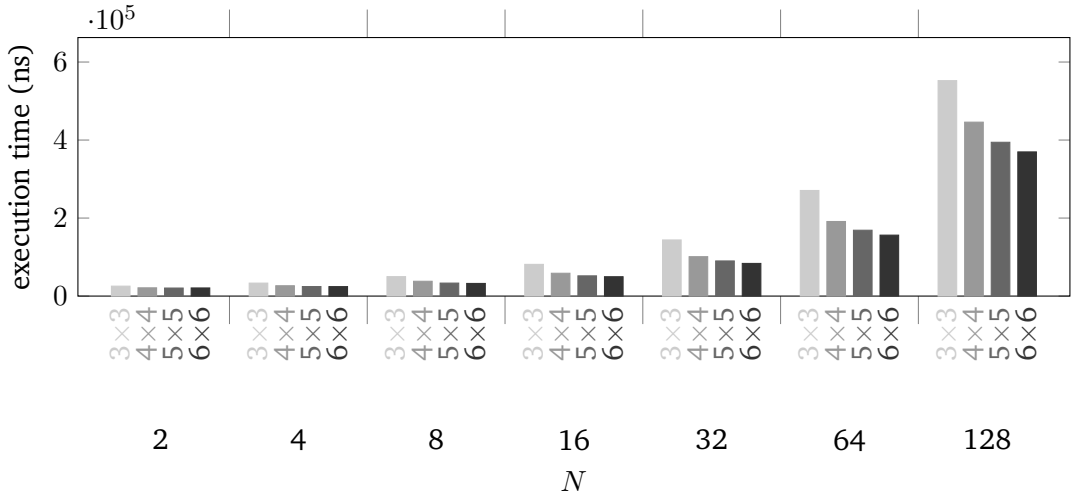


Figure 5.5.: Runtime comparison of $C_{60,60} = A_{60,N} \cdot B_{N,60}$ (float) including reconfiguration time with loop unrolling

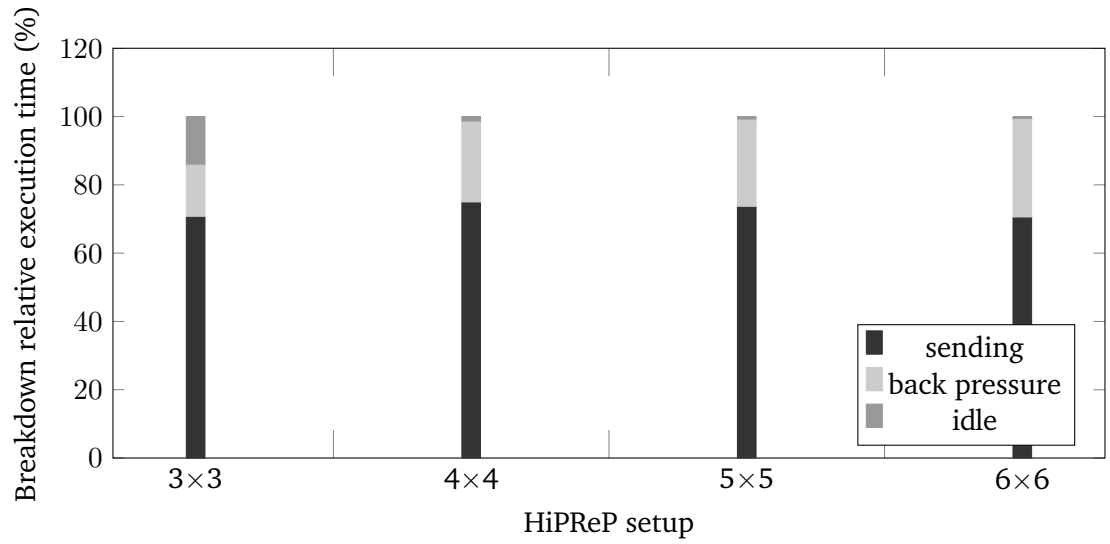


Figure 5.6.: Breakdown of accumulated execution time $\sum^{N} C_{60,60} = A_{60,N} \cdot B_{N,60}, N \in \{2, 4, 8, 16, 32, 64, 128\}$ (float) including reconfiguration time with loop unrolling; relative reconfiguration times: (3×3: 1.85 %, 4×4: 2.53 %, 5×5: 3.37 %, 6×6: 4.27 %)

Table 5.2.: Integer

CGRA Size	config	total	`rdcycle`	speedup
RISC-V only			48950	
compiled 4×4	1127	2972	8248	5.93
manual 4×4	676	3225	3663	13.18

Table 5.3.: floating point

CGRA Size	config	total	`rdcycle`	speedup
RISC-V only			58282	
compiled 4×4	1123	3524	9046	6.44
manual 4×4	659	3545	3937	14.80

on the 4×4 HiPReP setup of the manually mapped MMM with the results of the compiler mapped MMM.

In table 5.2 and table 5.3 the results of both mappings are shown. Columns "config", "total", and "`rdcycle`" are given in cycles. "total" is the number of cycles extracted from the log file, "`rdcycle`" is the number of cycles as read from Rocket's according control-status register (CSR) which also includes the number of cycles to patch the base addresses of HiPReP's AGUs in between reconfigurations. "speedup" is the factor by which the HiPReP performs the calculation faster than on Rocket.

Both compiler and manually mapped MMMs perform the MMM faster but the manually mapped MMM runs faster than the compiled one. This is due to the higher number of reconfigurations of the compiled MMM than required in the manually mapped MMM, since the manually mapped one can use the capability of the AGUs to iterate over two nested loops.

Regarding "total", the manually mapped MMM takes longer than the compiled one. This is due to the extensive patching in the compiled application which already loads the configuration in the L1D. The manually mapped MMM encounters a colder L1D and hence needs more time for the execution. Still, in this case, the compiled application needs to reconfigure the CGRA twice as often as the manually mapped one. Regarding "config", the floating point version of the application mappings can be configured faster than the integer MMM because in the floating point version we can use `MACC`, where we need to use two instructions, `MUL` and `ADD`, in the integer version.

In summary, the better AGU utilization and simpler patching in between reconfigurations make the manually mapped application faster than the compiled one.

In appendix B, the performance of a subset of compiled Polybench benchmarks is analyzed and in appendix C, the performance of other compiled benchmarks are given.

5.1.4. Optimization flow: Analysis

To optimize the HiPReP concept as described in section 2.3, we first need to analyze the current design with an evaluation methodology, and decide on an appropriate improvement technique. In the following, we motivate the choice of the roofline model as the evaluation methodology, apply it, and make a suggestion for an improvement technique which is task level parallelism. The application of task level parallelism is described in chapter 5.2.

Evaluation methodology: Roofline model

The MMM is well understood [38] and is an ideal candidate to sweep over a range of computational intensities to generate a roofline model by changing the matrices' dimensions. This visualizes how well a HiPReP setup can utilize the memory bandwidth or reach the theoretical peak performance, and whether a specific computational intensity is memory or compute bound. In particular, we sweep over different Ns for $C_{M,K} = A_{M,N} \cdot B_{N,K}$ for different HiPReP setups to explore different computational intensities. Once we know how the MMM is bound by the system, we can apply techniques from section 2.3.2 to improve the performance.

Computational roof of MMM The computational roof π of one $r \times c$ HiPReP per RoT on t RoTs running a MMM at frequency f is:

$$\pi = \frac{2 \cdot r \text{ Flop} \cdot c \text{ Flop} \cdot f \cdot t}{r \text{ Flop} + c \text{ Flop}} \tag{5.1}$$

This formula expresses that after loading $r + c$ values from memory (denominator), $r \cdot c$ multiplications and $r \cdot c$ additions are performed at frequency f (numerator). For quadratic HiPReP setups ($r \equiv c$), the formula can be simplified to:

$$\pi = r \text{ Flop} \cdot f \cdot t \tag{5.2}$$

In general the following formulas apply to compute the computational intensity *ci* for the MMM:

$$\#\text{data movement} = \left(\underbrace{M \cdot K}_{C_{M,K}} + \underbrace{M \cdot N}_{A_{M,N}} + \underbrace{N \cdot K}_{B_{N,K}} \right) \cdot \underbrace{4\,\text{B}}_{\#\text{ bytes for single precision floating point numbers}} \tag{5.3}$$

$$\#\text{Flop} = M \cdot K \cdot (\underbrace{N}_{\text{multiplications}} + \underbrace{N-1}_{\text{additions}})\text{Flop} \tag{5.4}$$

$$\text{ci} = \frac{\#\text{Flop}}{\#\text{data movement}} = \frac{M \cdot K \cdot (2 \cdot N - 1)\,\text{Flop}}{(M \cdot K + M \cdot N + N \cdot K) \cdot 4\ B} \tag{5.5}$$

For this experiment, a $C_{\#\text{RoT}\cdot 60,60} = A_{\#\text{RoT}\cdot 60,N} \cdot B_{N,60}$ is computed where N is scaled to explore different cis. From equation 5.5, we can compute the maximum ci we can reach with $M = \#\text{RoT} \cdot \text{M}' = 1 \cdot 60$ and $K = 60$ for a single RoT while running this MMM:

$$\lim_{N\to\infty} (\text{ci}) = \frac{M \cdot K\ \text{Flop}}{(M+K) \cdot 2\ \text{B}} = \frac{\#\text{RoT} \cdot \text{M}' \cdot K\ \text{Flop}}{(\#\text{RoT} \cdot \text{M}' + K) \cdot 2\ \text{B}} \tag{5.6}$$

$$\overset{\#\text{RoT}=1,\text{M}'=60,K=60}{=} 15\frac{\text{Flop}}{\text{B}} \tag{5.7}$$

We will refer to this maximum as *max ci*. To clarify, the *max ci* is a property of the application, not of the hardware.

Plotting the roofline model The roofline model of a single RoT with a HiPReP is plotted in figure 5.7. The blue line is the I/O bandwidth roof defined by the bandwidth of one DDR3 DRAM channel. The dashed, purple line shows the bandwidth ceiling as imposed by the RoT frequency for 6×6. The horizontal purple-lime line represents the computational ceiling π for 6×6 which is $\pi = r\ \text{Flop} \cdot f \cdot t = 6\ \text{Flop} \cdot 666\ \text{MHz} \cdot 1 = 3996\ \text{MFlop/s}$ (equation (5.2)). The measured Flop/s are below the π line and bandwidth ceiling as expected. The lines connect the measured Flop/s of one HiPReP setup: We can see that bigger HiPReP setups perform better than smaller ones, still the performance gain between the setups decreases such that the performance difference between 5×5 and 6×6 is barely noticable. Looking back at figure 5.6, this effect can be explained by the increasing back pressure for bigger setups. The peak performances are summarized in table 5.4. Independent of the HiPReP setup, these MMMs are I/O bandwidth bound for low cis and compute bound for higher cis.

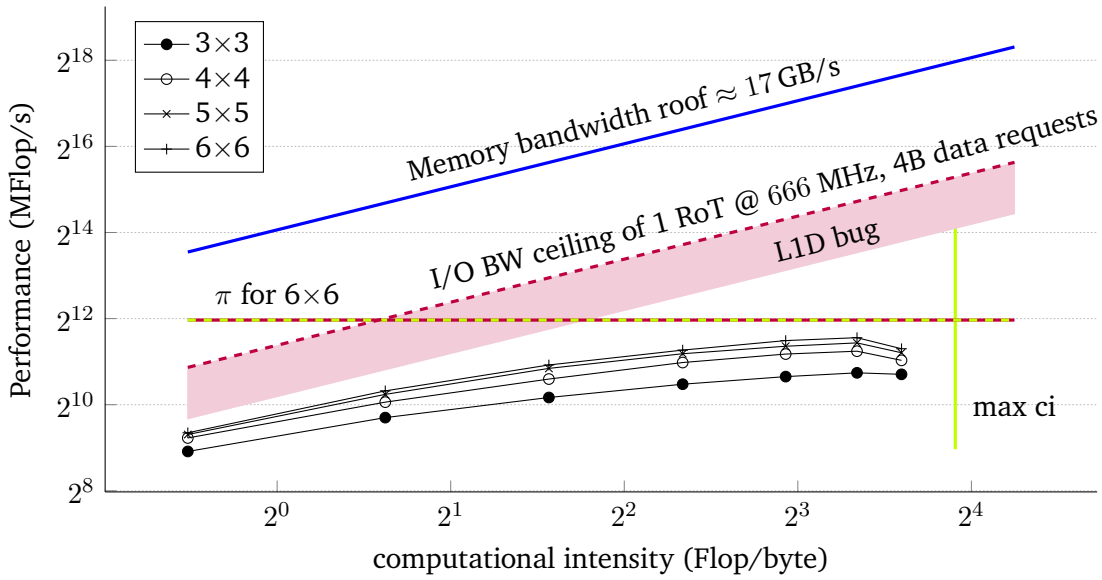


Figure 5.7.: Roofline model for MMM running on 1 RoT (*L1D bug* → appendix F.3) color-encoded dependencies:
blue: process node of memory
purple: process node of tile (UMC 65 LL)
lime: application mapping

The max ci is depicted as a lime vertical line in this plot to illustrate the upper ci bound imposed by the MMM, but is omitted in the following roofline model plots as it is irrelevant for scaling HiPReP hardware.

The *L1D bug* area is explained in the paragraph "Mind the gap" when we choose an improvement technique.

Roof dependencies The colors are further chosen to express that changing the respective dependency would influence the bounds. For instance, changing the process node from UMC 65 LL to a smaller process would lift the I/O bandwidth ceiling and the peak performance for any application. Plotting a roofline model for a different application would change the peak performance and the optional max ci. In other words, the peak performance depends on process node, hardware implementation and software application

Table 5.4.: Measured peak performances of the HiPReP setups on one RoT

HiPReP setup	peak performance (GFlop/s)
3×3	1.711
4×4	2.428
5×5	2.776
6×6	2.997

mapping. Using a different memory like DDR4 would push the memory bandwidth line further up.

Choosing an improvement technique

Mind the gap The first thing which catches the eye when evaluating the roofline model in figure 5.7 is the gap (pink area) between the I/O bandwidth ceiling (diagonal dashed purple line) and the achieved performance of the HiPReP setups in the memory bound area. It calls for immediate investigation: As an HPC accelerator, the HiPReP should not be a factor of 2.6 lower than the I/O bandwidth in the memory bound area but rather close to it.

In appendix F.3, we analyze why the HiPREPs do not reach this ceiling in detail, and conclude that the L1D's implementation has a bug, resulting in unnecessary reloads of data which have already been loaded before when the data set is bigger than 4 KiB, despite the L1D having more capacity. As a proof that this experiment is affected by the bug, we calculate the work data set size: The data set size is the accumulated memory footprint of the three matrices:

$$\begin{aligned}\text{size}_{\text{MMM}}(N) &= (M \cdot N + N \cdot K + M \cdot K) \cdot 4\ \text{B} \\ &\overset{\#\text{RoT}=1,\text{M}'=60,K=60}{=} (480\ N + 14400)\ \text{B}\end{aligned}$$

which means that for this experiment, we sweep over the range from 15360 B to 75840 B or 15 KiB to 74 KiB which is greater than 4 KiB - hence, the experiment is affected by the bug as described in the appendix, since the L1D has a capacity of 32 KiB.

Fixing this bug is out of the scope of this work and we accept this behaviour "as is" for now. We will refer to this bug as *L1D bug* in the following, thus the label in figure 5.7.

Task parallelism To improve the achieved performance, we explore a system with multiple RoTs with a HiPReP each in section 5.2, i.e. we apply task parallelism. Task parallelism will raise the bandwidth ceiling imposed by the RoT frequency we get with UMC 65nm LL technology as well as the computational roof. With $\frac{17\ \mathrm{GB/s}}{666\ \mathrm{MHz} \cdot 4\ \mathrm{B}} = 6.38 \approx 7$ RoTs, the combined I/O bandwidth ceiling theoretically crosses the memory bandwidth roof for 6×6 HiPReP setups which means, I/O bandwidth ceiling's impact becomes irrelevant. The relative distance between the measured performance of the HiPReP setups and the memory bandwidth roof at computational intensity 0.7 is $\frac{11900\ \mathrm{MFlop/s}}{665\ \mathrm{MFlop/s}} \approx 17.89$ as can be seen in figure 5.7. Considering this distance, and assuming linear speedup for adding more RoTs, we estimate that at least 18 RoTs are needed for the MMM to fully utilize the bandwidth of one DDR3 RAM channel, unless the L1D bug (appendix F.3) is fixed by the Rocket chip maintainers.

This technique is further explored in section 5.2 as an improvement technique and to demonstrate the HPC capability of HiPReP.

Memory Level Parallelism For completeness, we explore another improvement technique - storing matrix B in column-major ordering - in appendix E.1 which is a pure software technique to improve the MLP by accessing B's data sequentially in memory. We show that this technique's impact is negligible on the overall performance of the HiPReP setups running MMMs.

5.2. Multiple Rocket Tiles

So far, we concluded that an MMM running on a HiPReP on a single RoT is limited by the I/O bandwidth ceiling (actually the L1D bug at the moment, see appendix F.3) and cannot make use of the available memory bandwidth. Furthermore, strong and weak scaling need to be addressed to draw a conclusion on the HPC capability of HiPReP. In this section, we explore how multiple RoTs improve the overall performance of MMMs and whether HiPReP setups on multiple RoTs scale strongly and weakly after we show how the roofline models for different numbers of RoTs and PEs look like.

We will denote pairs of number of RoTs and HiPReP setup with parentheses in the following to express that there is a HiPReP with said setup on all RoTs, e.g. (1, 3×3) for one 3×3 setup on one RoT. In the experiments, the (8, 6×6) case and cases with more RoTs are omitted: This experiment results in an assertion being triggered on the system bus after

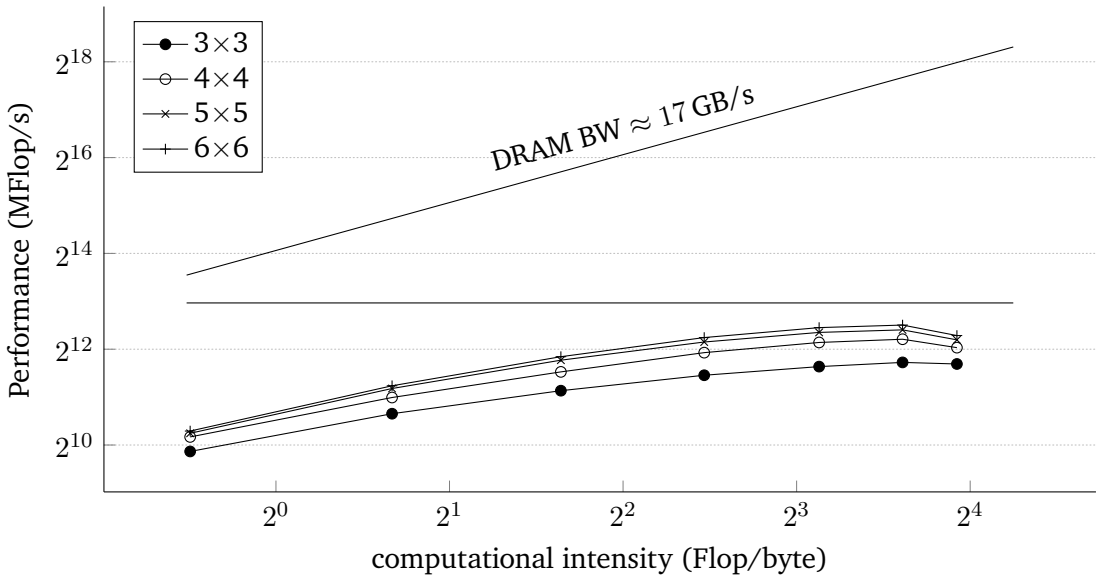


Figure 5.8.: Roofline model for 2 RoTs

some time, and debugging this is beyond of the scope of this work. This is further analyzed in section F.4. In the following, we refer to this bug as the *assertion bug*.

First, we will pick up where we left in section 5.1.4 and explore the roofline model in section 5.2.1 and how we can make full use of the available memory bandwidth. Then, we continue with strong and weak scaling experiments in the sections 5.2.2 and 5.2.3 to project the viability of multiple RoTs and to evaluate how efficient this scaling approach is compared to scaling the number of PEs on one RoT.

5.2.1. Roofline model

In the figures 5.8, 5.9, and 5.10, the rooflines of select HiPReP setups on two, four and eight RoTs are depicted, respectively. They illustrate the effect of increasing the number of RoTs, i.e. more task parallelism, on different setups when comparing the figures. The horizontal line for the peak performance always relates to the 6×6 setup - the peak performances for the other setups are omitted to ease the understanding but they are lower than the 6×6's peak performance.

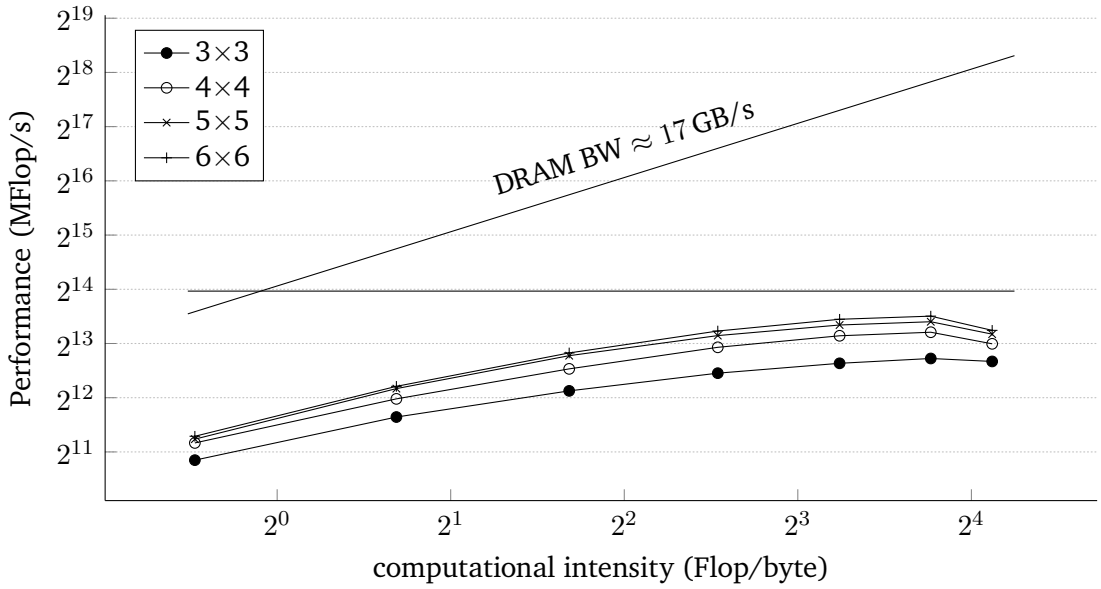


Figure 5.9.: Roofline model for 4 RoTs

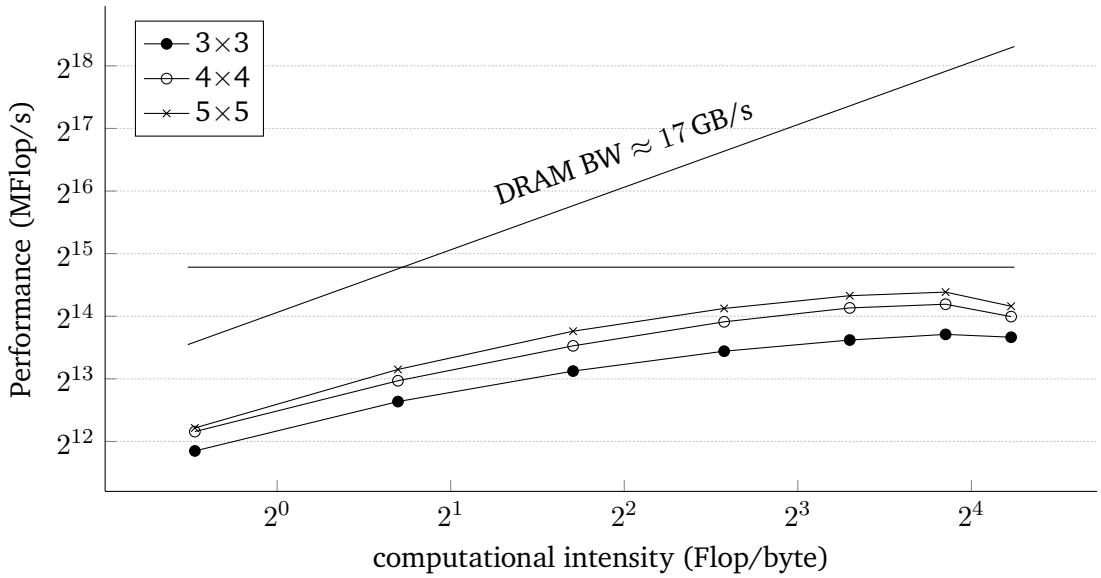


Figure 5.10.: Roofline model for 8 RoTs

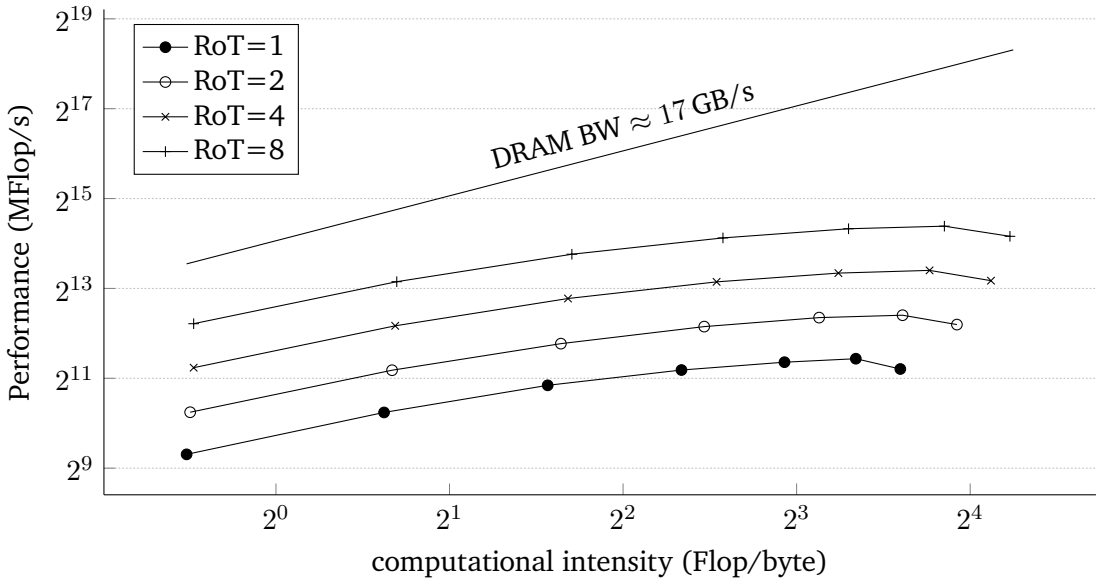


Figure 5.11.: Roofline model for 5×5 setups

As an example, figure 5.11 shows in one graph the roofline of 5×5 setups on different number of RoTs to visualize the effect of more RoTs on the performance.

At computational intensity 0.74, (8, 5×5) reaches 4771 MFlop/s. The memory bandwidth roof (one memory channel) at the same computational intensity is 12580 MFlop/s. This means that (8, 5×5) is $\frac{12580\ \mathrm{MFlop/s}}{4771\ \mathrm{MFlop/s}} \approx 2.6$ times below the memory bandwidth roof. We estimate that $8\ \mathrm{RoT} \cdot 2.6 = 20.8\ \mathrm{RoT} \approx 21$ RoTs with 5×5 setups could get close to the memory bandwidth roof, i.e. 525 PEs in total. This is a more reliable estimate than 18 RoTs from section 5.1.4 since we extrapolate from a measurement with more RoTs. A four channel DDR3 RAM's bandwidth could be theoretically fully utilized with $4 \cdot 21\ \mathrm{RoT} = 84$ RoTs, i.e. 2100 PEs in total, under the same assumptions. With the L1D bug (appendix F.3) resolved, we assume that we need fewer RoTs to fully utilize the memory bandwidth.

For the reasons explained in section F.4, we cannot simulate such big systems and leave this for future work. In summary, more RoTs are an effective technique to fully utilize the memory bandwidth and relieve the effects of the I/O bandwidth ceiling.

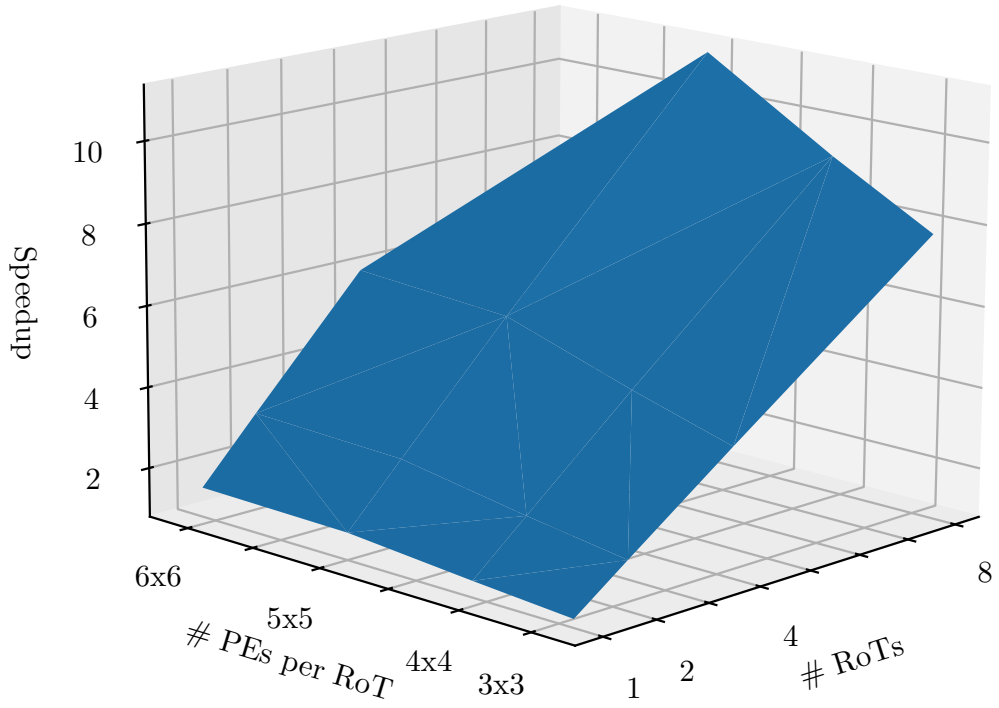


Figure 5.12.: Strong scaling of $C_{M,K} = A_{M,N} \cdot B_{N,K}$ with $M = K = 480, N = 64$

5.2.2. Strong scaling of MMM

The strong scaling of hardware systems is explored by running the same application while scaling a feature of a hardware system. The features to be scaled are the number of PEs per RoT and the number of RoTs for this experiment. Both span a two dimensional design space. The goal is to identify how much each of these parameters affects the total runtime for the same operation.

In figure 5.12, a $C_{M,K} = A_{M,N} \cdot B_{N,K}$ with $M = K = 480, N = 64$ application is run for 3×3, 4×4, 5×5, and 6×6 HiPReP setups, and 1, 2, 4, and 8 RoTs (and all combinations). The surface's nodes are the speedups relating to measured runtimes for the corresponding pair consisting of number of RoTs and number of PEs per RoT. The product of such a pair yields the total number of PEs. The speedups baseline is the (1, 3×3) setup.

The decrease in runtime when increasing the number of RoTs is bigger than the decrease in runtime when increasing the number of PEs per RoT, since the slope in # RoTs direction is steeper than the slope in # PEs per RoT direction. When comparing (1, 3×3) with (8, 3×3) (runtime(1, 3×3) = 21.02 ms, runtime(8, 3×3) = 1.888 ms), for instance, a speedup factor of 7.78 is attained which is close to 8. As each RoT can access the memory concurrently to the other RoTs, we conclude that the eight RoTs

1. can better harness the available memory bandwidth of the system than more PEs on one RoT and
2. are less prone to back pressure.

This adds up with the conclusion from section 5.1.4 where we see that the I/O bandwidth ceiling for one RoT is lower than the memory roofline, and only around 28 or more RoTs can utilize the available memory bandwidth with their concurrent memory requests according in theory.

In summary, it is an important insight for improving performance that increasing the number of RoTs scales better than increasing the number of PEs per RoT for applications similar to MMM. For a valid trade-off, the increased area cost for the additional RoTs and, hence, host processors, L1s, and FEs need to be taken into account.

5.2.3. Weak scaling of MMM

Weak scaling concerns the question how the runtime varies when an improvement technique is scaled at the same rate as the problem size. Ideally, the runtime should be the

same, no matter by which factor the parallelism and problem size are scaled. In other terms, this means that the scaled speedup should ideally be the same as the applied scaling factor. In the ideal case, the PEs are used efficiently by the application and mapping.

In figure 5.13, the efficiency of an MMM according to section 2.3.1 is shown in which each PE accumulates 64 products (multiply-accumulate). We use the two features "number of PEs per RoT" and "number of RoTs" like in the previous section. The input matrices A and B, as well as the output matrix C are scaled such that a $C_{\#\mathrm{RoT}\cdot r,c} = A_{\#\mathrm{RoT}\cdot r,64} \cdot B_{64,c}$ is computed, depending on the point in the design space. For $t(1)$ in the efficiency formula, (1, 1×1).

The evaluated efficiencies from the experiment are lower than the ideal efficiency for weak scaling which is 1 for this MMM. However, when only scaling the number of RoTs and keeping the number of PEs constant the efficiency degrades less than the other way around.

In conclusion, scaling HiPReP setups is more efficient when scaling the number of RoTs than scaling the number of PEs.

5.3. Comparison with NVidia's GH100

In section 2.2.2, we identified a total peak performance of 51.2 TFlops and a FP32 unit utilization of $88.6\ \%$ to compete with the performance of NVidia's GH100. HiPReP's (8, 5×5) peak performance is 21.825 GFlops, i.e. the GH100 is

$$\frac{\pi_{\mathrm{GH100}}}{\pi_{(8,5\times5)}} = \frac{51.2\ \mathrm{TFlops}}{21.825\ \mathrm{GFlops}} \approx 2346 \tag{5.8}$$

times faster. The utilization of HiPReP's PEs is calculated with

$$\mathrm{utilization}_{PE} = \frac{21.825\ \mathrm{GFlops}}{8\ \mathrm{RoT} \cdot (5 \cdot 5)\ \frac{\mathrm{PE}}{\mathrm{RoT}} \cdot 2\ \frac{\mathrm{Flop}}{\mathrm{PE}} \cdot 704\ \mathrm{MHz}} \approx 0.078 \tag{5.9}$$

which is $\frac{\mathrm{utilization}_{FP32}}{\mathrm{utilization}_{PE}} = \frac{0.886}{0.078} = 11.36$ times less than the FP32 utilization on the GH100 architecture. For reasoning about the performance gap, the process node, warming up caches, and the impact of custom FMAs are discussed next.

The superiority of the GH100 in absolute numbers, as shown in equation 5.8, can be partly explained by the differing process nodes: The GH100 is produced with a custom

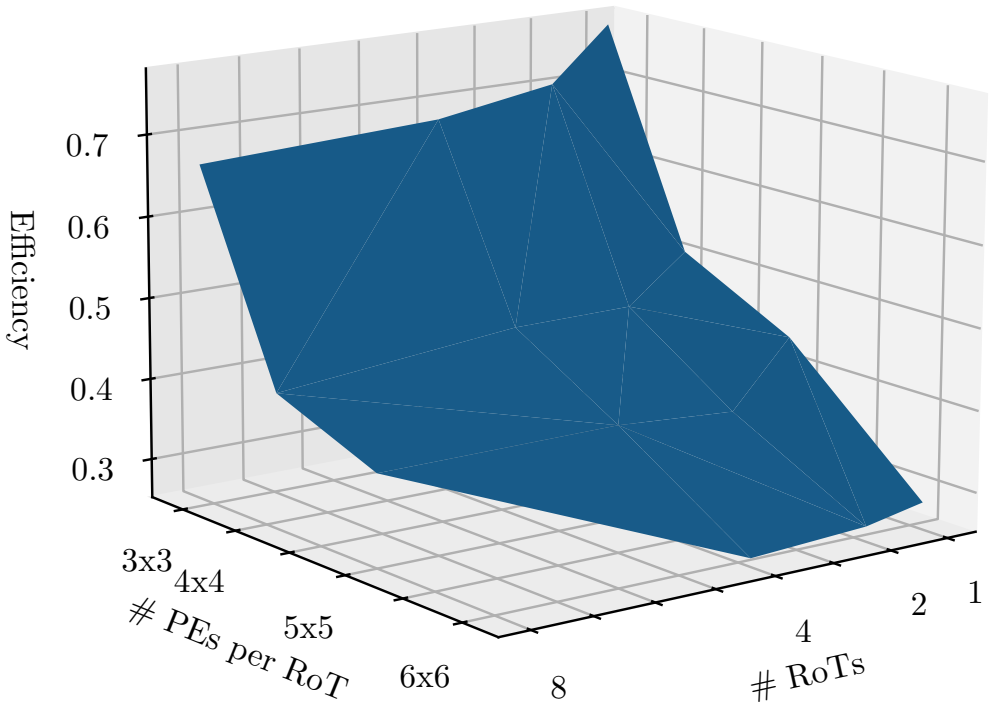


Figure 5.13.: Efficiency

TSMC 4N process while HiPReP synthesized in the UMC65LL process. This difference contributes to the

$$\frac{f_{\text{GH100}}}{f_{(8,5\times5)}} = \frac{1980\ \text{GHz}}{704\ \text{MHz}} = 2.815 \tag{5.10}$$

times higher clock frequency of the GH100. Further, the GH100 features

$$\#\ \text{FP32} = 144\ \text{SM} \cdot 4\ \frac{\text{TC}}{\text{SM}} \cdot 32\ \frac{\text{FP32}}{\text{TC}} = 18432\ \text{FP32} \tag{5.11}$$

units and the (8, 5×5) HiPReP only

$$\#\ \text{PE} = 8\ \text{RoT} \cdot (5 \cdot 5)\ \frac{\text{PE}}{\text{RoT}} = 200\ \text{PEs} \tag{5.12}$$

(and implicitly FMAs), which explains another factor of

$$\frac{\#\ \text{FP32}}{\#\ \text{PE}} \approx 92.16. \tag{5.13}$$

Normalizing equation 5.8 to the clock frequencies and number of FPUs, and under the assumption that HiPReP's FMAs are similar to GH100's FP32, the peak performance gap shrinks to $\frac{2346}{2.815 \cdot 92.16} = 9.04\times$ per floating point unit.

Another important aspect impacting the performance is the unknown experimental setup from NVidia: First, we do not know with which application and under which prerequisites GH100's peak performance is measured, i.e. whether the caches were warmed up, for instance. The (8, 5×5) HiPReP's performance is measured with cold caches. Second, the L1D bug (appendix F.3) degrades HiPReP's performance. Once this bug is fixed, HiPReP's performance needs to be reevaluated with warmed-up caches. The performance per PE is expected to improve and their utilization to increase. The according experiment is left to future work.

Lastly, FMA unit designs are highly customized [57]. It is fair to assume that the GH100 is manufactured with custom FMA units available as design blocks during synthesis. The Synopsys Design Compiler synthesizing HiPReP in the UMC 65nm LL process does not have custom FMA units available. Instead, it synthesizes HiPReP's FMA units from common standard cells which are not as optimized as custom blocks in terms of performance and area. HiPReP's FMAs feature ten pipeline stages to push up the overall clock frequency whereas custom FMAs reach high frequencies with typically three stages. For a fair comparison, the HiPReP needs to be synthesized with a more competitive, custom FMA unit.

6. Conclusion

In this work, the HPC capability of CGRAs is explored with the example of HiPReP as a RoCC accelerator in Chipyard. The HiPReP architecture shows promise when it comes to scaling, making it interesting to explore further in the future, despite it being limited by bugs discovered in Chipyard framework. Provided these bugs get fixed, the evaluation methodology presented in this work can be reused to update the results.

The methodology is presented, first, while explaining the iterative optimization steps and by evaluating the proposed system for HPC in general. Second, the microarchitecture of the HiPReP CGRA and the host system integration is described. Third, the HiPReP CGRA is evaluated with the roofline model, as well as strong and weak scaling experiments.

During the evaluation of the single RoT system, it became apparent that such a system is limited by the I/O bandwidth ceiling imposed by the PDK UMC 65 LL. In the process, the L1D bug is identified which further reduces the attainable bandwidth of RoCC accelerators in general, and HiPReP in particular. But even without the L1D bug, due to the I/O bandwidth ceiling, one RoT synthesized in UMC 65 LL cannot utilize the available memory bandwidth of a single DDR3 RAM channel.

Designing systems with multiple RoTs is proven to be a promising technique to utilize the available memory bandwidth and to cross the I/O bandwidth ceiling of one RoT. Yet, these experiments expose the assertion bug which restricts the number of RoTs which can be simulated in one system. The collected data of the successful runs are conclusive to state that more RoTs can utilize the bandwidth. In combination with using a newer process node and/or fixing the L1D bug, the number of RoTs could be smaller than extrapolated during the analysis. This needs to be shown in future work.

In terms of HPC capability, first, a strong scaling experiment is conducted to evaluate how the total performance can be effectively increased: whether by scaling up the number of PEs of HiPReP, scaling up the number of RoTs, or even a combination of both features. The major insight is that scaling up the number of PEs per RoT is, on the one hand, limited

by the I/O bandwidth ceiling and, on the other hand, the application mapping on one HiPReP. Scaling up the number of RoTs is therefor more effective to increase performance but comes at a higher area cost. Second, the weak scaling experiment reveals that PEs are more efficient when they are spread over multiple RoTs rather than putting them on one single RoT.

In summary, a CGRA like HiPReP is a promising accelerator for HPC applications like the MMM. Currently, the 5×5 setup appears to be the most efficient setup to utilize the available memory bandwidth on a RoT, unless the L1D bug is fixed. Other applications might not harness all PEs of the CGRA all the time which might require bigger setups. Further, CHiPReP's development should also focus on how to split applications into sub-tasks such that they can be mapped and managed on multiple RoTs equipped with HiPReps, as this yields the best gains for performance. To compete with state-of-the-art GPUs, HiPReP setups must be synthesized in a similar process node, while CHiPReP's maturity progresses.

7. Future work

Simulation of a bigger system Based on this work, we see the need for simulating a bigger system to further explore a multiple RoT system with HiPREPs. But as explained in section 5.2, Chipyard is facing issues at a certain system size (assertion bug, appendix F.4), in addition to the L1D bug (appendix F.3). A way to side-step this problem and even speedup simulation time, might be to reimplement HiPReP on a higher abstraction level and simulate these systems in SST[1], for instance, to also reduce the simulation time.

Simulating a bigger system would enable fully utilizing the memory bandwidth further, and exploring strong scaling further with a higher impact of cache sizes and memory hierarchy properties, as well as network topologies.

Memory access of packed data In a 64b architecture, loads and stores from CPU to L1D happen in 8B words. Since HiPReP is targeting single precision floats, i.e. 4B words, two memory accesses to consecutive, 8B aligned memory addresses can be packed into one memory access. The detection of packable memory accesses could be done either by hardware, e.g. by the FE, or ahead of time by the compiler to configure the AGUs accordingly.

Memory access patterns The AGUs used in this work generate addresses in the direct memory access pattern. In the future, other access patterns (i.e. indirect and dynamic, according to the definition by Abbaszadeh et al. [28]) need to be supported for less regular applications.

[1] Structural Simulation Toolkit, https://github.com/sstsimulator

Compiler Once the compiler development is done, we want to benchmark a HiPReP system with an HPC benchmark suite. For that, the compiler should support the mapping of an application on multiple RoTs and fully exploit the AGU capability to handle two nested loops.

Newer process node Using a newer process node based on FinFET (or GAAFET/-nanowire, nanosheet, CFET [58]) for the hardware synthesis will likely improve the performance of the HiPReP by sheer power and frequency scaling. Then, we could apply the methodologies presented in this work again to tune the number of PEs per HiPReP and number of RoTs/HiPREPs.

A. Ready-valid protocol

The ready-valid protocol is a handshake protocol which ensures that data is transmitted reliably between two components, i.e. from (data) producer to (data) consumer. In combination with FIFO buffers (first in, first out buffers), the protocol offers a solution to the producer-consumer problem. The problem was first described by Dijkstra [59]. It concerns how data is sent from one component (producer) to other components (consumers) through communication channels and how to make sure that the consumer never reads from an empty buffer and the producer never writes to a full buffer. A channel would be wires in our case, accompanied by a ready and a valid wire between the producer and the consumer.

In figures A.1a and A.1b, two diagrams are shown with a full and a simplified representation of two blocks using the ready-valid handshake protocol. Both representations being functionally equivalent, the full version represents the actual interface more realistically, while the simplified version reduces the drawing overhead and enhances the readability. In figure A.1c, a FIFO is added which is functionally invisible to both endpoints. Throughout this document, the simplified version is used in block diagrams.

Whenever a producing component is sending valid, and the consuming component sends ready, both ends of this communication channel can safely assume that the data is successfully transmitted. If the producer is not sending valid and the consumer is ready, the consumer needs to wait. If the producer has valid data and the consumer is not ready, the producer needs to wait. The latter case is called back pressure. If the producer is not valid and the consumer is not ready, nothing will happen.

By embedding FIFO buffers (or short FIFOs) in the channel, the probability of back pressure can be reduced. The theory behind this is formally laid out in *queueing theory* [60]. A FIFO provides two ready-valid interfaces, one to the producer, and one to the consumer. Internally, the FIFO manages a small memory or buffer. Whenever the producer has valid data and the FIFO has a free entry in the buffer, the FIFO will accept the data and send ready to the producer. Whenever the FIFO has a valid entry, it will send valid to

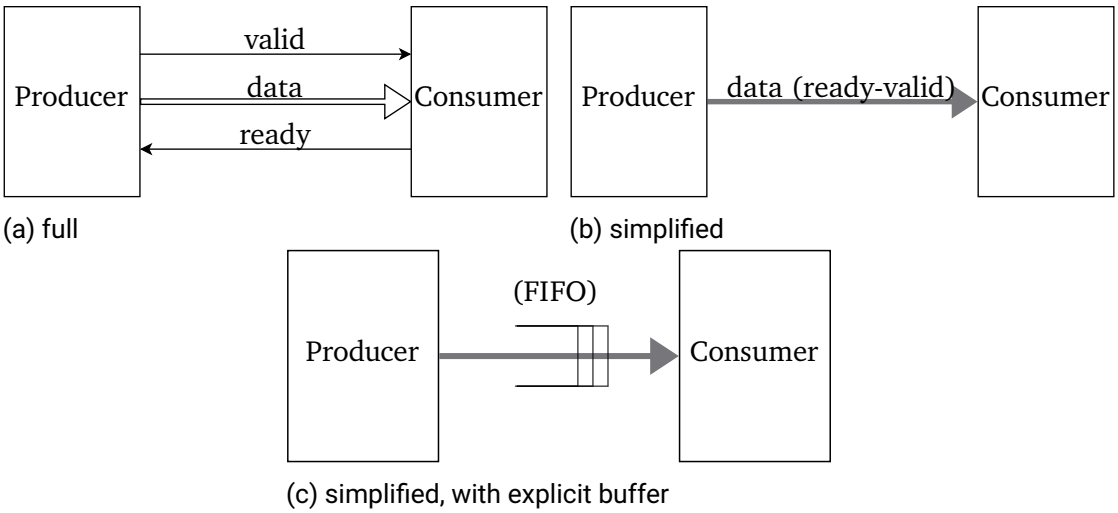


Figure A.1.: Block diagram of a producer and a consumer using the ready-valid handshake protocol

the consumer. Whenever the consumer is ready, and the FIFO has a valid entry in the buffer, the FIFO will invalidate the entry. Still, if the FIFO is full and the consumer is still not ready, the FIFO will also give back pressure to the producer. Queueing theory aims to characterize the data traffic and determines the required capacity of the FIFOs with respect of the service time of the various endpoints to, first, provide an optimal throughput in complex networks and, second, to prevent deadlocks. In figure A.1c, a FIFO buffer is explicitly depicted, but in general, FIFO buffers are omitted in the block diagrams since they are functionally invisible. In Chisel, a FIFO is an intrinsic type, called "Queue", while the ready-valid protocol is handled by Chisel's "Decoupled" interface.

To illustrate the importance of this protocol, we will analyze the HiPReP integration, first, with respect to the memory subsystem, and second, with respect to the PEs in the CGRA.

A.1. Example: Memory subsystem

In the case of HiPReP, we consider the following endpoints on a higher level during the processing phase: The memory subsystem, the AGUs, and the CGRA, with the FE

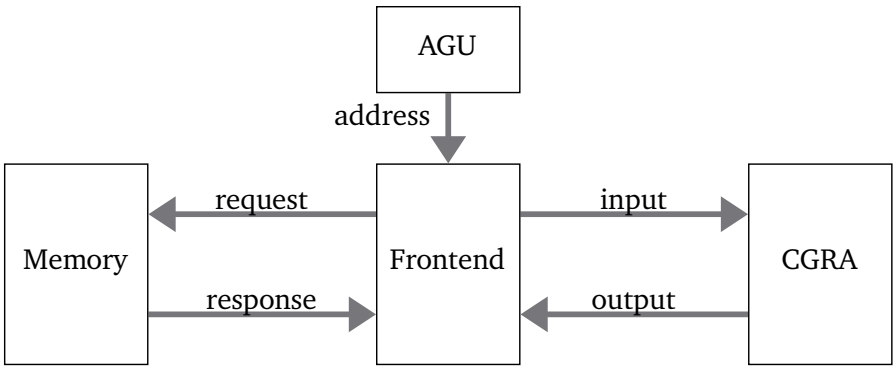


Figure A.2.: High-level block diagram of the HiPReP during the processing phase

Table A.1.: Message sequence in HiPReP

no.	message type	producer	consumer
1.	address	AGU	FE
2.	read request	FE	memory
3.	read response	memory	FE
4.	input data	FE	CGRA
5.	output data	CGRA	FE
6.	write request	FE	memory
7.	write response	memory	FE

connecting all, as can be seen in figure A.2. In general, ordering the endpoints from short to long service time, the

AGUs can always produce addresses (i.e. short service time, generating high arrival rate at FE).

CGRA has varying service times, depending on the contexts. We consider the best-case for dimensioning the queues, while the functionality guaranteeing functionality.

memory subsystem has varying service times between three to hundreds of cycles due cache hits and misses. We consider the worst-case, i.e. which is always miss (long service time).

The FE handles these endpoints such that the data is efficiently delivered. The producers and consumers of the different message types in this system are shown in table A.1.

The FE also acts as a buffer for the memory and CGRA at the same time, and gives back pressure to the AGU when necessary. On the "input" links, FIFOs are inserted to even out variations of service times and to prevent deadlocks. The CGRA service times vary depending on the application and mapping. For the remaining links, no buffers are placed. The FE also never gives back pressure on the "response" link from the memory subsystem, as making the memory, as the slowest component, wait is considered counter productive for high performance by design. The FE rather throttles the address generation ahead of time by pressuring back on the AGUs when it is about to run out of buffer space, i.e. entries in the outstanding request table.

A.2. Example: CGRA

In the case of HiPReP's CGRA, the PEs and FE are connected according to the diagram in figure 3.5. For this analysis, we consider every PE a producer *and* consumer. The service time of each PE might differ between one and more cycles, limited by the number of context memory entries and depending on the application and mapping. For a short service time, one word is output every cycle, for a long service time, it requires all contexts in the context memory to be run to produce one word. For this reason, FIFOs are placed on all links in-between the PEs to prevent deadlocks when dealing with complex mappings.

B. Polybench

The compiler for HiPReP, CHiPReP, is an on-going effort to efficiently map applications onto HiPReP setups. In this section, preliminary results from a subset of Polybench 4.2.1 benchmarks[1] [53] are presented. An in-depth study about the acceleration of Polybench kernels on HiPReP has been conducted prior by Weinhardt [61].

The L1D bug also applies to these experiments (appendix F.3). The CHiPReP does not optimize loops as of now. For instance, by unrolling loops more parallelism is to be exposed in the future which we expect will have an beneficial impact on the performance. But for now, the results show the "bare" speedup without any loop transformation.

As baselines, the Rocket core is used and the speedup is based on that. In table B.1, the results of running the benchmarks on Polybench's mini data set, and in table B.2 the results for the small data set are assembled. The unit of the columns "RV only" and "HiPReP 3×3" is cycle.

For the mini data set, only for "atax", "bicg", and "gemver" actual speedups can be reported, while for the small data set, for all kernels except "correlation" and "jacobi-2d" speedups are achieved. For these kernels, it roughly pays off to run compiled mappings on a 3×3 setup when data sets as big as 128 KB or bigger need to be processed. This adds up with the conclusion of section 5.2.3, where it can be seen that the HiPReps are the more efficient, the more data one configuration processes. As CHiPReP does not unroll loops as of now, the speedups are mostly due to the concurrent address generation by the AGUs and data processing by the PEs.

[1] github.com/MatthiasJReisinger/PolyBenchC-4.2.1

Table B.1.: MINI_DATASET (16 KB) speedups

Kernel	RV only [cycle]	HiPReP 3×3 [cycle]	Speedup
gemm	102652	213250	0.48
correlation	180479	1278800	0.14
covariance	175219	254015	0.69
2mm	91040	307980	0.3
3mm	152615	422967	0.36
atax	41361	39828	1.04
bicg	43204	32385	1.33
mvt	25724	53172	0.48
gemver	89767	71599	1.25
gesummv	18070	19562	0.92
symm	183011	331387	0.55
syrk	168180	238103	0.71
durbin	30894	39811	0.78
ludcmp	430773	686860	0.63
jacobi-1d	14456	15725	0.92
jacobi-2d	929850	1556072	0.6

Table B.2.: SMALL_DATASET (128 KB) speedups

Kernel	RV only [cycle]	HiPReP 3×3 [cycle]	Speedup
gemm	5926786	2776603	2.13
correlation	6627883	11473194	0.58
covariance	6502217	5347169	1.22
2mm	5046219	4589506	1.1
3mm	8878203	7493933	1.18
atax	455906	261121	1.75
bicg	427842	209828	2.04
mvt	521841	393481	1.33
gemver	917400	544948	1.68
gesummv	185627	123021	1.51
symm	4699128	4371788	1.07
syrk	3408237	2481848	1.37
durbin	247299	194576	1.27
ludcmp	11317886	10987931	1.03
jacobi-1d	199391	52895	3.77
jacobi-2d	21913540	26539899	0.83

C. Other compiled benchmarks

In this section, results of computing a finite impulse response filter (FIR) (tables C.1 and C.2) and a two dimensional convolution filter (tables C.3 and C.4) are presented. As baseline for the speedups, the same experiment is run on the Rocket core. The benchmarks are compiled by the CHiPReP.

For the larger HiPReP setups, each DFG node/operation is mapped to its own PE, allowing a fully pipelined implementation. Due to the experimental state of the compiler, some numbers in the columns showing the number of L1 accesses are colored red to indicate that the compiler is generating mappings with a sub-optimal number of memory requests, i.e. it requests the same data multiple times from different AGUs, which constrains the potential performance. A few runs, however, use mappings with a constrained number of AGUs but this is an experimental feature at the moment. This avoids the use of additional AGUs and the unnecessary memory read accesses. These runs are marked with "Lim. AGU" in the tables. Column "cyc./L1 acc." shows how much overhead due to cache loading/writing and pipeline filling occurs compared to the optimal pipeline ($\approx$ 1 cyc./L1 acc.).

The numbers show that for small work loads like in these benchmarks, first, the overhead of larger contexts for bigger HiPReP setups diminishes the speedup, and second, the I/O bandwidth ceiling limits the speedup.

Table C.1.: FIR filter: Integer

setup	config [cycle]	total [cycle]	duration [cycle]	speedup	L1 accesses	cyc./L1 acc.	R/W
RISC-V only			23828				
4×4	343	3144	3948	6.04	993	2.82	1/1
6×6	498	3231	3944	6.04	1493	1.83	2/1
8×8	632	3422	4243	5.62	1493	1.87	2/1

Table C.2.: FIR filter: floating point

setup	config [cycle]	total [cycle]	duration [cycle]	speedup	L1 accesses	cyc./L1 acc.	R/W
RISC-V only			23359				
4×4	404	3161	4446	6.6	993	2.78	1/1
6×6	503	3272	4559	6.44	1493	1.85	2/1
8×8	605	3284	4597	6.39	1493	1.79	2/1

Table C.3.: 2D convolution filter: Integer

setup	config [cycle]	total [cycle]	duration [cycle]	speedup	L1 accesses	cyc./L1 acc.	R/W
RISC-V only	-	-	9647				
2×2	452	3004	4969	1.94	1341	1.9	2/1
3×3	597	3942	6237	1.55	2241	1.49	4/1
4×4	630	4104	6383	1.51	2241	1.55	4/1
4×4 Lim. AGU	589	3203	5144	1.88	1341	1.95	2/1

Table C.4.: 2D convolution filter: floating point

setup	config [cycle]	total [cycle]	duration [cycle]	speedup	L1 accesses	cyc./L1 acc.	R/W
RISC-V only	-	-	27375				
4×4	1426	7871	12739	2.15	2384	2.7	5/1
6×6	1652	6165	11087	2.47	2784	1.62	6/1
6×6 Lim. AGU	1474	5593	9826	2.79	1584	2.6	3/1
8×8	1710	6046	10859	2.52	2384	1.82	5/1
8×8 Lim. AGU	1664	6112	10758	2.54	1984	2.24	4/1

D. PE instructions

The PE instructions or contexts are designed to cover fundamental mathematical operations on integers and floating point numbers. Integer operations can only be performed on signed numbers, and floating point numbers need to be single-precision (32 b) as of now.

Also, integer-float and float-integer conversions, as well as control flow, i.e. comparisons, logic operations and conditional branches, are supported.

In the remainder of this section, tables are given which summarize all currently implemented contexts.

Table D.1.: Instruction Set Architecture: R-Typ integer

Context	Latency	Annotation
ADD_INT	1	@Target = @Src1 + @Src2
SUB_INT	1	@Target = @Src1 - @Src2
MUL_INT	1	@Target = @Src1 * @Src2
LSL	1	@Target = @Src1 « @Src2
ASR	1	@Target = @Src1 » @Src2
LESS	1	@Target = @Src1 < @Src2
LEQ	1	@Target = @Src1 $\leq$ @Src2
GT	1	@Target = @Src1 > @Src2
EQ	1	@Target = @Src1 == @Src2
NEQ	1	@Target = @Src1 $\neq$ @Src2
AND	1	@Target = @Src1 & @Src2
OR	1	@Target = @Src1 | @Src2
XOR	1	@Target = @Src1 ^ @Src2

Table D.2.: Instruction Set Architecture: I-Type

Context	Latency	Annotation
FP_INT	1	Convert `float` to `int`
INT_FP	1	Convert `int` to `float`
ADDI_INT	1	@Target = @Src1 + Src2
SUBI_INT	1	@Target = @Src1 - Src2
MULI_INT	1	@Target = @Src1 * Src2
LSLI	1	@Target = @Src1 « Src2
ASRI	1	@Target = @Src1 » Src2
LESI_INT	1	@Target = @Src1 < Src2
LEQI_INT	1	@Target = @Src1 $\leq$ Src2
GEQI_INT	1	@Target = @Src1 $\geq$ Src2
GRI_INT	1	@Target = @Src1 > Src2
EQI_INT	1	@Target = @Src1 == Src2
NEQI_INT	1	@Target = @Src1 $\neq$ Src2
ANDI	1	@Target = @Src1 & Src2
ORI	1	@Target = @Src1 | Src2
XORI	1	@Target = @Src1 ^ Src2

Table D.3.: Instruction Set Architecture: R-Type float

Context	Latency	Annotation
ADD_FP	10	like ADD_INT, but for `float`
SUB_FP	10	like SUB_INT, but for `float`
MUL_FP	10	like MUL_INT, but for `float`
FMA	10	@Target = @RS * @Src1 + @Src2
MACC	10	@RS+= @Src1 * @Src2
FIMA	10	@Target = @Src2 - @Src1 * @RS
FMS	10	@Target = @RS * @Src1 - @Src2

RS := Special Register, implicitly used by ternary instructions; currently it is R31

Table D.4.: Instruction Set Architecture: control

Context	Latency	Annotation
BNEZ	1	Jump to Target, if @Src1 $\neq$ 0
BEZ	1	Jump to Target, if @Src1 == 0
JUMP	1	Jump to Target
NOP	1	No operation
SET_MAX_PC	1	Jump to Target, whenever PC == Src1
END	1	Stop processing and notify the FE's control unit

E. Mapping improvement of manually mapped MMM

E.1. B in column-major order

When computing $C = AB$, B's entries are accessed column-wise while B is stored in row-major order which means that potentially not all bytes of a cacheline are used before it is replaced in the cache. A way to remedy this is to store B in column-major order, i.e. accessing B's columns sequentially in the memory. But how much can we gain from this computing $C = AB$ on HiPReP?

In figure E.1, two roofline models are plotted, one with row- and one with column major ordering. We can see that the gain from row-major ordering is mostly negligible for arithmetic intensities below 8, which is implicitly 64 rows in B. For higher arithmetic intensities, we can see a small improvement.

Overall, the impact of changing the store ordering of B is negligible mostly, as the HiPReP performs close to equally well with both orderings.

E.2. Warming up the caches

When evaluating the performance of a CPU core or hardware accelerator, benchmarks might face cold caches as the simulator system "just started up". The benchmark is evaluated in a worst-case scenario where all data needs to be loaded from memory. Depending on the system, the application is then memory bandwidth or memory latency bound.

To evaluate the corresponding best-case corner, i.e. the data is already in the cache because of a previous task, for instance, the compulsory cache misses at the beginning need to be

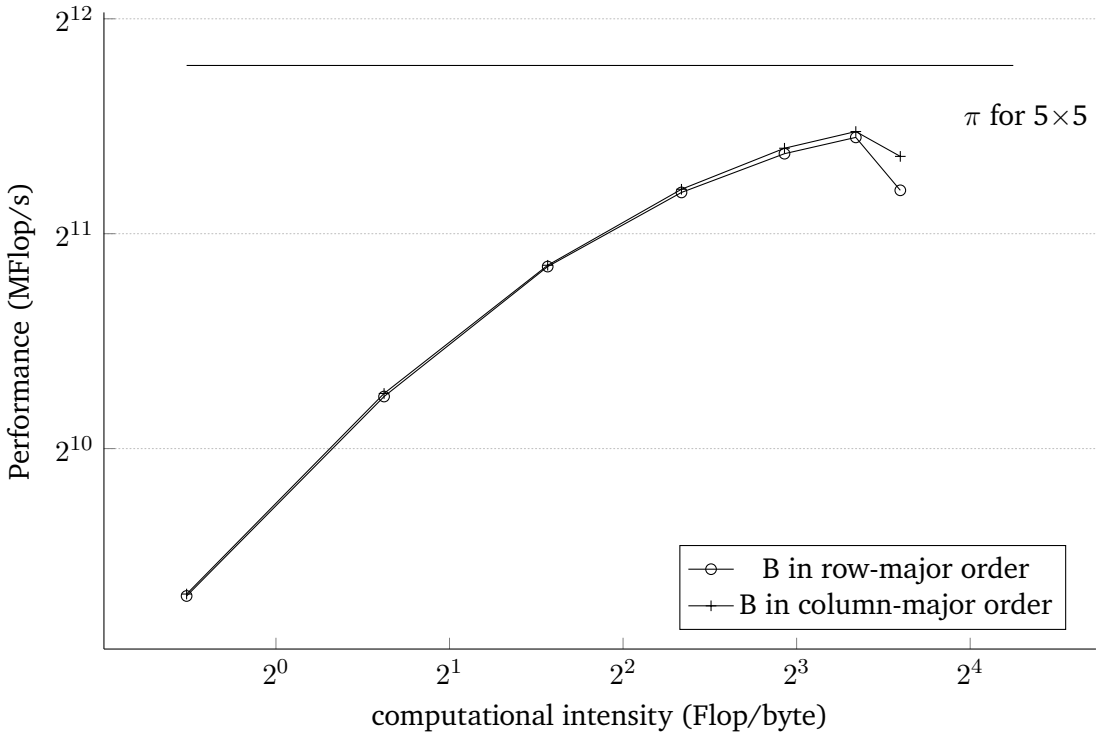


Figure E.1.: Roofline models for 5×5 setup showing the effect of the value ordering of B in $C = AB$

eliminated. Only then, the CPU or accelerator can be evaluated for its (theoretical) peak performance. For this reason, the caches need to be warmed up before the evaluation.

In general, caches can be warmed up through prefetch requests. A programmer or compiler usually inserts prefetch requests into the program to instruct the core/L1D to fetch data ahead of time to have it close when it might be used and to avoid unnecessary pipeline stalls. A prefetch can be achieved in two ways: cache prefetching and register prefetching [39, p. 92]. Cache prefetching is only loading the prefetch data into the L1D, register prefetching loads prefetch data into a CPU register.

Since gcc does not support `__builtin_prefetch` for RISC-V at this point [1], cache prefetching is not viable, and we need to resort to register prefetching to warm up the

[1] accessed July 16th, 2024: https://gcc.gnu.org/onlinedocs/gcc/RISC-V-Built-in-Functions.html

L1D. A code snippet for this is given in algorithm 10 in which the array `input` is loaded into the variable `preload`. Although `preload` is overwritten all the time, the associated values from `input` are expected to remain in L1D.

Algorithm 10 Warming up Rocket's L1D

```
1 unsigned input[] = { /* ... */ };
2 // ...
3 volatile unsigned preload;
4 for (unsigned i = 0; i < sizeof(input)/sizeof(input[0]); i++)
5     preload = input[i];
```

We take a close look on the effectiveness of warming up Rocket's L1Ds in section F.2, since this evaluation is also used for the bottleneck analysis and reveals the L1D bug.

F. Rocket Core bottleneck analysis

During the integration of HiPReP, bottlenecks of the system were analyzed to improve on the performance numbers. The found bottlenecks are part of the Rocket implementation and therefore outside this thesis' scope but affect its evaluation.

First, a bottleneck in the SimpleHellaCacheIF is identified and resolved in section F.1. Second, the experiment and results of warming up the Rocket's L1D are presented in section F.2 which leads to, third, the evidence of another performance bottleneck (L1D bug), described in section F.3. Fourth, we expose an implementation issue (assertion bug) in section F.4.

F.1. Replay queue in SimpleHellaCacheIF

For integration purposes of HiPReP into Chipyard, initially a program/configuration was designed which repeatedly accessed the same memory address. While analyzing the waveform, the SimpleHellaCacheIF was not always ready to accept a new memory request, although the addressed value should have been in the L1D after an initial miss. This can be observed at signal `io_mem_req_ready` in figure F.1, and in more detail in figure F.2: It is always high for two cycles and low for one cycle.

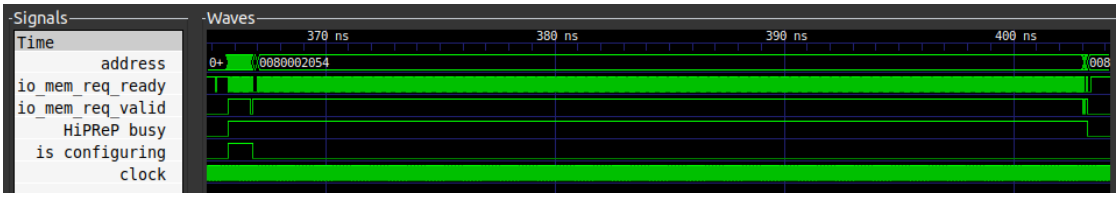


Figure F.1.: Waveform with replay queue depth = 2, overview

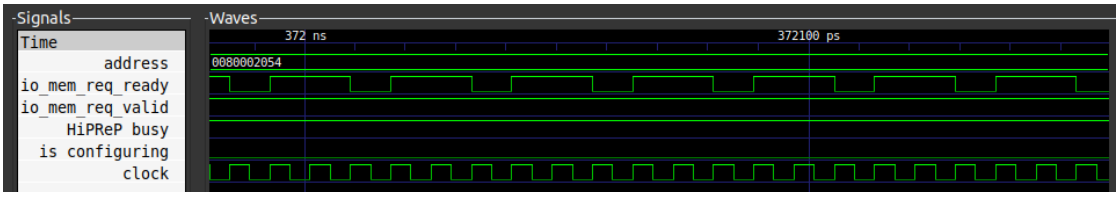


Figure F.2.: Waveform with replay queue depth = 2, detailed

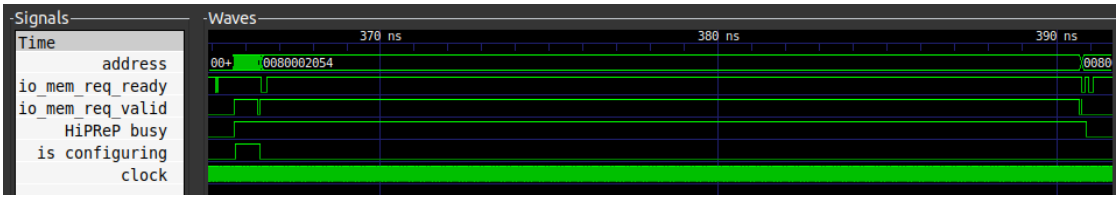


Figure F.3.: Waveform with replay queue depth = 3

After some investigation, the SimpleHellaCacheIF's replay queue was identified to cause the back pressure: Any request coming from a RoCC accelerator is sent to the L1D and queued in the replay queue concurrently. But the L1D might sent a not-acknowledge, i.e. *nack*, for the request from two cycles before. In such a case, the replay queue sends the request again, otherwise, i.e. no nack, it discards the request. But when there is a continuously sending RoCC accelerator like HiPReP, the two entries in the replay queue are not sufficient, as it cannot cover the latency three cycles from sending a request to the L1D to L1D nack. Hence, it needs to stall the new request from the RoCC accelerator as it doesn't know yet whether it can discard the oldest request. By changing the depth to three, this source of back pressure can be eliminated. A waveform from a run of the same program as before, but with three entries in the replay queue, is shown in figure F.3: The `io_mem_req_ready` is now continuously high. Detailed numbers are given in table F.1.

This issue was reported and the fix merged (https://github.com/chipsalliance/rocket-

Table F.1.: Comparison of different replay queue depths

replay queue depth	2	3
# back pressure	1575	22
# requests	3090	3090
# total cycles	4688	3148
back pressure ratio	33%	0%

chip/pull/3678).

F.2. Impact of warming up L1D

In section E.2, we proposed a way to warm up the L1D cache in a RISC-V system. In addition, we resolved a performance bottleneck by extending the replay queue in Rocket chip's SimpleHellaCacheIF (section F.1). In the following, we show that L1D warm up works and improves the performance. However, we still faced unexpected performance issues in terms of back pressure from L1D which we analyze in F.3.

Algorithm 11 Assembly from register prefetching

```
8000037c: 96818693  addi a3,gp,-1688 # 80011638 <preload>
80000380: 4398      lw   a4,0(a5)
80000382: c298      sw   a4,0(a3)
80000384: 0791      addi a5,a5,4
80000386: fec79de3  bne  a5,a2,80000380 <main+0x150>
```

We can confirm the setup and correct L1D cache size (`d-cache-size` = 32768 B) with the following section from the build log (algorithm 12), automatically generated during the construction of the system.

The experiment is designed to show two things:

1. register prefetching works (section F.2.1), and
2. warming up improves performance by eliminating L1D misses (section F.2.2).

As we will see in section F.3, preloading Rocket's L1D does not improve performance as much as expected.

F.2.1. Functionality

In the experiment, Rocket core is set up to preload data for the HiPReP to consume after. As baseline, the same program is run, but without preloading. As long as the data to

Algorithm 12 Build log

```
L26: cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        timebase-frequency = <1000000>;
        L8: cpu@0 {
                clock-frequency = <0>;
                compatible = "sifive,rocket0", "riscv";
                d-cache-block-size = <64>;
                d-cache-sets = <64>;
                d-cache-size = <32768>;
                d-tlb-sets = <1>;
                d-tlb-size = <32>;
                device_type = "cpu";
                hardware-exec-breakpoint-count = <1>;
                i-cache-block-size = <64>;
                i-cache-sets = <64>;
                i-cache-size = <16384>;
                i-tlb-sets = <1>;
                i-tlb-size = <32>;
                mmu-type = "riscv,sv39";
                next-level-cache = <&L5>;
                reg = <0x0>;
                riscv,isa = "rv64imafdc";
                riscv,pmpgranularity = <4>;
                riscv,pmpregions = <8>;
                status = "okay";
                timebase-frequency = <1000000>;
                tlb-split;
                L6: interrupt-controller {
                        #interrupt-cells = <1>;
                        compatible = "riscv,cpu-intc";
                        interrupt-controller;
                };
        };
};
```

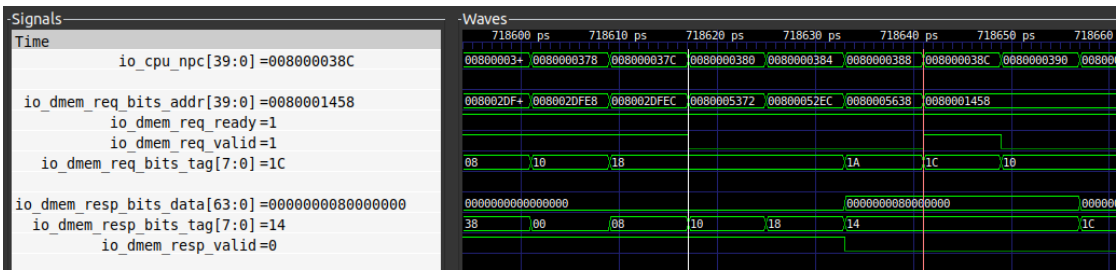


Figure F.4.: Waveform: first load of prefetch

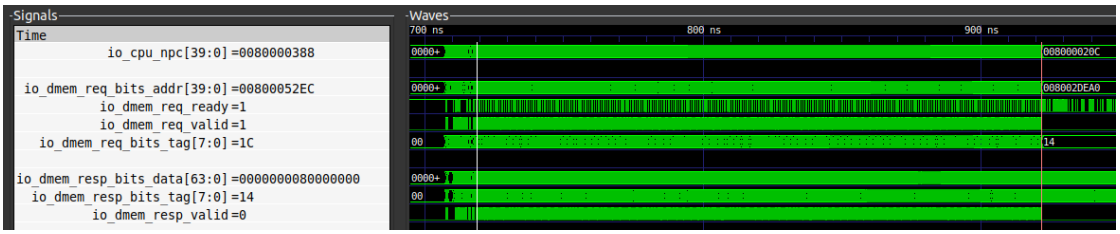


Figure F.5.: Waveform: last load of prefetch

preload fits into the L1D (32 KiB), we expect that the data can be accessed with no back pressure, resulting in a constant speedup. The preload assembly is shown in algorithm 11.

The `input` array starts at 0x80001458 and ends at 0x800052f0 in our example. This is 16 KiB together with HiPReP's contexts which fits into the 32 kB L1D. The related load instruction is located at address 0x80000380. This is confirmed in figure F.4: The first/left marker shows when the first prefetch is fetched from the Rocket's frontend (`io_cpu_npc`), the second marker shows when the Rocket's LSU issues the corresponding load to the first address (0x80001458). In figure F.5, the last marker shows when the last prefetch address (0x800052ec) is sent. Hence, all entries of `input` were loaded into variable `preload` and we conclude that all data has been preloaded into the L1D.

F.2.2. Performance improvement

For proving the performance improvement, we expect that the access time should be improved and no back pressure due to L1 cache misses, when the HiPReP accesses array `input`. This experiment is performed after changing the depth of the replay queue in the SimpleHellaCacheIF, as described in section F.1.

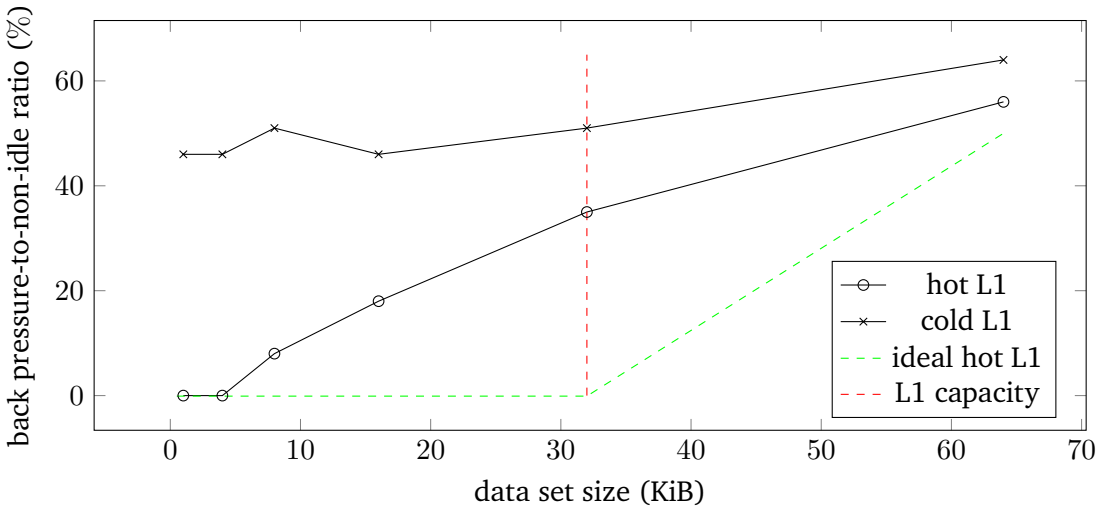


Figure F.6.: Sweep over data sizes to preload on Rocket's L1D

In figure F.6, we swept over a range of data set sizes which are preloaded to examine the back pressure-to-non-idle ratio on HiPReP while sending requests. By non-idle, we mean the cycles during which the HiPReP has a request to sent. The cold L1 behaves as expected: Due to the 64b architecture, on a `lw`, i.e. 32 b/4 B load, roughly half of the requests will miss in the beginning, and half will hit since half of the data is loaded into the L1. (This is because on a `lw` miss, an 8 B word (cache line size) is loaded into the L1 cache before serving the 4 B response, i.e. the missing word *and* another 32 b value are loaded. That "other" 32 b value is the value required next in this benchmark.) Looking at the hot L1, it can be observed that the back pressure rises, as more data is preloaded. Below 4 KiB, no back pressure is observed with a hot L1 cache. Due to the increasing back pressure beyond 4 KiB, the attainable speedup for warming up L1D is the more constrained, the more data is preloaded, as can be seen in figure F.6. The presence of back pressure contradicts our prediction for a warmed up L1D. This is investigated in section F.3.

In conclusion, warming up the L1D works and improves performance. Yet, contrary to our expectation, the back pressure increases with the size of the preloaded data set according to figure F.6. This reduces the attainable speedup, as plotted int figure F.7.

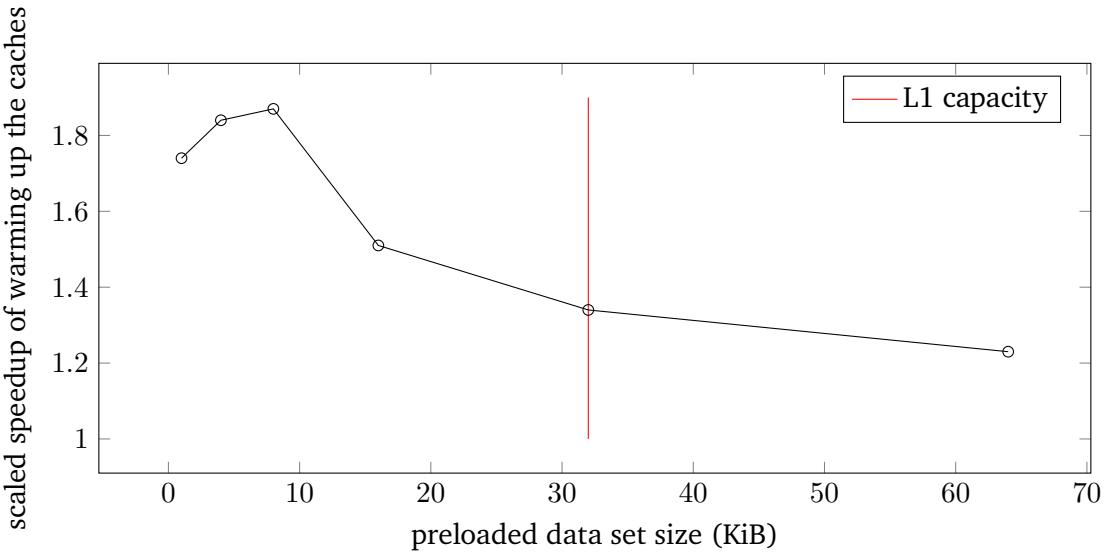


Figure F.7.: Effectiveness of warming up Rocket's L1D

F.3. Unexpected reloads in L1D

In section F.2.1, back pressure from the L1D for data set sizes greater than 4 KiB is identified despite the L1D being warmed up which we call *L1D bug*. We refer to the same experiment in the following. It is confirmed that even for data set sizes which fit entirely in the L1D, back pressure is given to the HiPReP which ultimately reduces the achievable bandwidth when the data set is bigger than 4 KiB. Ideally, back pressure should only be given from a cache when data is missing and the MSHR is full.

Furthermore, looking at the back pressure as a metric of cache misses, the hot L1D cache curve resembles the case for a 4 KiB L1D and a hot L2, despite we confirmed in the log that the L1D has a capacity of 32 KiB. This raises the question whether the cache is refilling lines which are already present.

To verify whether the L1D is refilling data which are wrongfully treated as missing, the response port from the tilelink is monitored: When the data set fits into the L1D, and is preloaded before the HiPReP starts executing, there should be no data coming from the L2 or memory on tilelink's port d [45, p. 17]. The corresponding waveform is shown in figure F.8.

After the last/right marker in the waveform, HiPReP begins to execute. This is indicated by "HiPReP idle" switching from high to low. During HiPReP's execution phase, "auto_out_d_valid" is switching repeatedly between high and low. This indicates that responses from L2 or memory are arriving with data. On "auto_out_d_bits_data", the value 0x0000000a0000000a is shown. This is the data in the input array of the benchmark. The L1D is filling cache lines which were already loaded before during the warm up phase. During the unnecessary refill, the L1D gives back pressure to the HiPReP, limiting the performance.

Further investigation shows that the respective cache lines falsely appear to be in "Nothing" state. This is one of the coherence states (Nothing, Branch, Trunk, Dirty) of the tilelink protocol. Our hypothesis is that the L1D cannot handle back-to-back requests well as issued by HiPReP. The back-to-back requests can be observed when looking at the "io_mem_req_valid" signal in figure F.8: It is constantly high during HiPReP's deployment. In contrast, Rocket core does not build up such a request pressure which can be seen at signal "io_dmem_req_valid" which is switching when it accesses memory.

A hypothesis for future debugging is that the L1D coherence protocol implementation is not maintaining the coherence state faithfully, rendering the L1D as being unaware that more capacity is available. In addition, the high request frequency from HiPReP may probe cache lines in a vulnerable window of the coherence protocol's implementation and might not be reached by a general purpose CPU like Rocket core. The design and conduction of the further experiments for debugging purposes are out of the scope of this work.

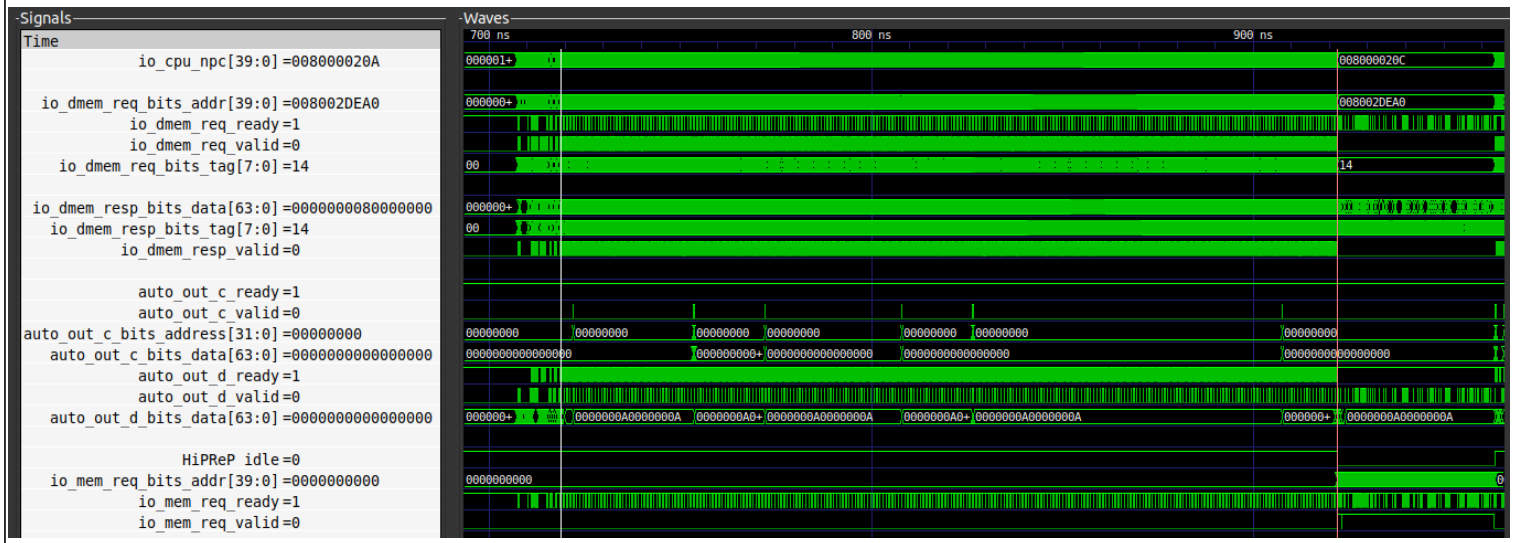


Figure F.8.: Waveform: L1D misses despite warm up

F.4. Assertion triggers with multiple RoTs

When simulating systems with multiple RoTs, the simulation fails with the following assertion being triggered:

```
Assertion failed in
TOP.TestHarness.chiptop.system.subsystem_sbus.fixer.monitor_8
```

Multiple RoTs equipped with HiPREPs build up a high request pressure on the tilelink. Such a request pressure is likely never build up with general purpose processors and, hence, hard to test with these setups. We assume that the HiPREPs trigger it as a consequence of the hypothesized implementation problem of the coherence protocol, as explained in section F.3, or another issue with the coherence protocol's implementation: The HiPREPs might be driving the global coherence state to a state which is not handled by the implementation, causing this assertion to trigger. Again, this needs to be investigated with further experiments which are out of the scope of this work.

Due to this assertion trigger, system simulations beyond 8 RoTs with 5x5 HiPREP setups fail and no numbers can be reported. This assertion trigger is spurious since the protocol should be able to handle this by not accepting more requests at least, i.e. back pressure, and not by firing an assertion.

Abbreviations

ADRES Architecture for dynamically reconfigurable embedded systems

AGU Address generator unit

ALU Arithmetic logic unit

AMIDAR Adaptive microinstruction driven architecture

assertion bug See appendix F.4

CDFG Control data flow graph

CGRA Coarse-grained reconfigurable array

CPU Central processing unit

CSR Control status register

DLP Data-level parallelism

DDR RAM Double data rate RAM

DFG Data flow graph

DRAM Dynamic random access memory

DSE Design space exploration

DSU Data streaming unit

DySER Dynamically specializing execution resources

ELF Executable and linkable format

FE HiPReP frontend

FIFO First in, first out buffer

FIR Finite impulse response

FMA Fused multiply-add unit (float)

GPU Graphics processing unit

hartid hardware thread identification number

HiPReP High performance reconfigurable processor

HPC High performance computing

ILP Instruction-level parallelism

ISA Instruction set architecture

L1 Level 1 cache

L1D Level 1 data cache

L1D bug See appendix F.3

L1I Level 1 instruction cache

L2 Level 2 cache

LSU Load-store unit

MIMD Multiple instruction, multiple data

MLP Memory-level parallelism

MMM (Dense) matrix-matrix multiplication

MRRG Module routing resource graph

PCU Pattern compute unit

PDK Process development kit

PE Processing element

PMU Pattern memory units

PTW Page table walker

RAM Random access memory

RISC Reduced instruction set computer architecture

RoCC Rocket custom coprocessor interface

RoT Rocket Tile

RSVP Reconfigurable streaming vector processor

SGMF Single-graph multiple flows

SIMD Singe instruction, multiple data

SM Streaming multiprocessor

SSE Streaming SIMD Extension

SoC System on chip

SRAM Static random access memory

TLB Translation look-aside buffer

TLP Thread-level parallelism

VLIW Very long instruction word processor

List of Figures

Bibliography

[1] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974. DOI: 10.1109/JSSC.1974.1050511.

[2] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th annual international symposium on Computer architecture*, 2011, pp. 365–376.

[3] F. Bouwens, M. Berekovic, B. De Sutter, and G. Gaydadjiev, "Architecture enhancements for the adres coarse-grained reconfigurable array," in *High Performance Embedded Architectures and Compilers: Third International Conference, HiPEAC 2008, Göteborg, Sweden, January 27-29, 2008. Proceedings 3*, Springer, 2008, pp. 66–81.

[4] C. Kim, M. Chung, Y. Cho, M. Konijnenburg, S. Ryu, and J. Kim, "Ulp-srp: Ultra low-power samsung reconfigurable processor for biomedical applications," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 3, pp. 1–15, 2014.

[5] S. Venkataramani et al., "Rapid: Ai accelerator for ultra-low precision training and inference," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2021, pp. 153–166.

[6] Z. Ebrahimi and A. Kumar, "Green: An approximate simd/mimd cgra for energy-efficient processing at the edge," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 10, pp. 2874–2887, 2024. DOI: 10.1109/TCAD.2024.3383349.

[7] E. Strohmaier, J. Dongarra, H. Simon, and M. Meurer, "Top 500," Tech. Rep., 2024, online; accessed 07-January-2025.

[8] Y. Fu, E. Wu, A. Sirasao, S. Attia, K. Khan, and R. Wittig, "White paper: UltraScale and UltraScale+ FPGAs. Deep Learning with INT8 Optimization on Xilinx Devices," Xilinx, Inc., Tech. Rep., 2017, online; accessed 20-January-2025.

[9] I. SambaNova Systems, "White paper: Accelerated Computing with a Reconfigurable Dataflow Architecture," Tech. Rep., 2021, online; accessed 20-January-2025.

[10] L. Liu et al., "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–39, 2019.

[11] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[12] P. Nannipieri et al., "Vlsi design of advanced-features aes cryptoprocessor in the framework of the european processor initiative," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 2, pp. 177–186, 2021.

[13] B. D. Sutter, P. Raghavan, and A. Lambrechts, "Coarse-grained reconfigurable array architectures," *Handbook of signal processing systems*, pp. 427–472, 2019.

[14] M. Wijtvliet, L. Waeijen, and H. Corporaal, "Coarse grained reconfigurable architectures in the past 25 years: Overview and classification," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, IEEE, 2016, pp. 235–244.

[15] Z. Li, D. Wijerathne, and T. Mitra, "Coarse-grained reconfigurable array (cgra)," *Handbook of Computer Architecture*, pp. 1–41, 2022.

[16] M. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966. DOI: 10.1109/PROC.1966.5273.

[17] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972. DOI: 10.1109/TC.1972.5009071.

[18] M. J. Flynn and K. W. Rudd, "Parallel architectures," *ACM computing surveys (CSUR)*, vol. 28, no. 1, pp. 67–70, 1996.

[19] V. Govindaraju et al., "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, 2012.

[20] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Design and evaluation of dynamically specialized datapaths with the dyser architecture," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 2010.

[21] V. Govindaraju, T. Nowatzki, and K. Sankaralingam, "Breaking simd shackles with an exposed flexible microarchitecture and the access execute pdg," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, IEEE, 2013, pp. 341–351.

[22] C.-H. Hoy et al., "Performance evaluation of a dyser fpga prototype system spanning the compiler, microarchitecture, and hardware implementation," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2015, pp. 203–214.

[23] B. Mei, A. Lambrechts, J.-Y. Mignolet, D. Verkest, and R. Lauwereins, "Architecture exploration for a reconfigurable architecture template," *IEEE Design & Test of Computers*, vol. 22, no. 2, pp. 90–101, 2005.

[24] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev, "Architectural exploration of the adres coarse-grained reconfigurable array," in *Reconfigurable Computing: Architectures, Tools and Applications: Third International Workshop, ARC 2007, Mangaratiba, Brazil, March 27-29, 2007. Proceedings 3*, Springer, 2007, pp. 1–13.

[25] NVIDIA, "Nvidia's next generation cuda compute architecture," Tech. Rep., 2009. [Online]. Available: https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[26] D. Voitsechov and Y. Etsion, "Single-graph multiple flows: Energy efficient design alternative for gpgpus," *ACM SIGARCH computer architecture news*, vol. 42, no. 3, pp. 205–216, 2014.

[27] R. Prabhakar et al., "Plasticine: A reconfigurable architecture for parallel paterns," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 389–402, 2017.

[28] M. Abbaszadeh, T. S. Abdelrahman, R. Azimi, T. S. Czajkowski, and M. Goudarzi, "Efficient data streaming for a tightly-coupled coarse-grained reconfigurable array," in *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2023, pp. 435–443. DOI: 10.1109/IPDPSW59300.2023.00075.

[29] S. Ciricescu et al., "The reconfigurable streaming vector processor (rsvp/spl trade/)," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, IEEE, 2003, pp. 141–150.

[30] S. Gatzka and C. Hochberger, "The amidar class of reconfigurable processors," *the Journal of Supercomputing*, vol. 32, pp. 163–181, 2005.

[31] D. L. Wolf, L. J. Jung, T. Ruschke, C. Li, and C. Hochberger, "Amidar project: Lessons learned in 15 years of researching adaptive processors," in *2018 13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2018, pp. 1–8. DOI: 10.1109/ReCoSoC.2018.8449384.

[32] N. Corporation, "Nvidia h100 tensor core gpu architecture," Tech. Rep., 2023.

[33] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483–485.

[34] J. L. Gustafson, "Reevaluating amdahl's law," *Commun. ACM*, vol. 31, no. 5, pp. 532–533, May 1988, ISSN: 0001-0782. DOI: 10.1145/42411.42415. [Online]. Available: https://doi.org/10.1145/42411.42415.

[35] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009, ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. [Online]. Available: https://doi.org/10.1145/1498765.1498785.

[36] G. E. P. Box, "Science and statistics," *Journal of the American Statistical Association*, vol. 71, no. 356, pp. 791–799, 1976. DOI: 10.1080/01621459.1976.10480949. eprint: https://www.tandfonline.com/doi/pdf/10.1080/01621459.1976.10480949. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/01621459.1976.10480949.

[37] S. Ristov, R. Prodan, M. Gusev, and K. Skala, "Superlinear speedup in hpc systems: Why and when?" In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2016, pp. 889–898.

[38] D. Ernst, G. Hager, J. Thies, and G. Wellein, "Performance engineering for real and complex tall & skinny matrix multiplication kernels on gpus," *The International Journal of High Performance Computing Applications*, vol. 35, no. 1, pp. 5–19, 2021. DOI: 10.1177/1094342020965661. eprint: https://doi.org/10.1177/1094342020965661. [Online]. Available: https://doi.org/10.1177/1094342020965661.

[39] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011, ISBN: 012383872X.

[40] V. C. Cabezas and M. Püschel, "Extending the roofline model: Bottleneck analysis with microarchitectural constraints," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, 2014, pp. 222–231. DOI: 10.1109/IISWC.2014.6983061.

[41] A. Amid et al., "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020, ISSN: 1937-4143. DOI: 10.1109/MM.2020.2996616.

[42] J. Bachrach et al., "Chisel: Constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, 2012, pp. 1216–1225.

[43] K. Asanović et al., "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr. 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[44] A. Rao, "The rocc doc v2: An introduction to the rocket custom coprocessor interface," University of California, San Diego, Tech. Rep. [Online]. Available: https://docs.google.com/document/d/1CH2ep4YcL%5C_ojsa3BVHEW-uwcKh1FlFTjH%5C_kg5v8bxVw.

[45] SiFive, "Sifive tilelink specification," SiFive Inc., Tech. Rep., 2023. [Online]. Available: https://sifive.cdn.prismic.io/sifive/928d6a82-77a9-4291-8b60-5e815429b1ab_tilelink_spec_1.9.3.pdf.

[46] P. S. Käsgen, M. Weinhardt, and C. Hochberger, "Dynamic scheduling of pipelined functional units in coarse-grained reconfigurable array elements," in *Architecture of Computing Systems–ARCS 2019: 32nd International Conference, Copenhagen, Denmark, May 20–23, 2019, Proceedings 32*, Springer, 2019, pp. 156–167.

[47] P. S. Käsgen, M. Weinhardt, and C. Hochberger, "A coarse-grained reconfigurable array for high-performance computing applications," in *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2018, pp. 1–4. DOI: 10.1109/RECONFIG.2018.8641720.

[48] J. H. et al., "Berkeley hardware floating-point units," Tech. Rep., 2023.

[49] F. Schuiki, F. Zaruba, T. Hoefler, and L. Benini, "Stream semantic registers: A lightweight risc-v isa extension achieving full compute utilization in single-issue cores," *IEEE Transactions on Computers*, vol. 70, no. 2, pp. 212–227, 2021. DOI: 10.1109/TC.2020.2987314.

[50] A. López-Lagunas and S. M. Chai, "Memory bandwidth optimization through stream descriptors," in *Proceedings of the 2005 workshop on MEmory performance: DEaling with Applications, systems and architecture*, 2005, pp. 57–64.

[51] P. Käsgen, M. Messelka, and M. Weinhardt, "Hiprep: High-performance reconfigurable processor - architecture and compiler," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 380–381. DOI: 10.1109/FPL53798.2021.00074.

[52] M. Weinhardt, M. Messelka, and P. Käsgen, "Chiprep—a compiler for the hiprep high-performance reconfigurable processor," *Electronics*, vol. 10, no. 21, p. 2590, 2021.

[53] T. Yuki and L.-N. Pouchet, "Polybench 4.2. 1 (pre-release)," Tech. Rep., 2016.

[54] A. Podobas, K. Sano, and S. Matsuoka, "A survey on coarse-grained reconfigurable architectures from a performance perspective," *IEEE Access*, vol. 8, pp. 146 719–146 743, 2020. DOI: 10.1109/access.2020.3012084. [Online]. Available: https://doi.org/10.1109%5C%2Faccess.2020.3012084.

[55] "Sifive fu540-c000 manual v1p0," SiFive, Inc., Tech. Rep., 2018. [Online]. Available: https://d2pn104n81t9m2.cloudfront.net/documentation/chips/freedom-u540-c000-manual/.

[56] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," in *Fourth Workshop on Computer Architecture Research with RISC-V*, vol. 5, 2020, pp. 1–7.

[57] E. Quinnell, E. E. Swartzlander, and C. Lemonds, "Floating-point fused multiply-add architectures," in *2007 Conference Record of the Forty-First Asilomar Conference on Signals, Systems and Computers*, IEEE, 2007, pp. 331–337.

[58] J. Ryckaert et al., "Enabling sub-5nm cmos technology scaling thinner and taller!" In *2019 IEEE International Electron Devices Meeting (IEDM)*, 2019, pp. 29.4.1–29.4.4. DOI: 10.1109/IEDM19573.2019.8993631.

[59] E. W. Dijkstra, "My recollections of operating system design," Tech. Rep., 2000.

[60] M. Harchol-Balter, *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.

[61] M. Weinhardt, "An analysis of mapping polybench kernels to hpc cgras," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2022, pp. 647–654.