Decoupled Spatial Architecture Framework

PolyArch Resesarch Team

Oct 19, 2022

CONTENTS:

DSAGEN is a framework for designing decoupled-spatial architectures, a class of programmable accelerators. DSAGEN makes use of a variety of tools for spatial-scheduling, compilation, simulation, ISA-generation, and hardware generation.

Important: New to DSAGEN? Jump to the *Setup* page for setup instructions.

CHAPTER

ONE

DSA FRAMEWORK BASICS

These sections will walk you through the basics of the DSA framework:

- First, we will overview the framework and its goals/capabilities.
- Next, we will go over the components of the framework.
- Finally, we describe how to setup the repository.

1.1 Setup

1.1.1 Prerequisites

We highly recommend you use Docker to setup the environment. By downloading this Dockerfile, you can simply setup the environment by typing to build a docker image:

```
$ sudo docker build . -t polyarch/dsa-framework:latest
```

Then you can start a docker container by executing command below. The docker option *-v /home/<user>/dsa-share:/root/dsa-share* allows you to share files between host machine and docker container.

```
$ sudo docker run -tid [-v /home/<your username>/dsa-share:/root/dsa-share] --
↪name=overgen polyarch/dsa-framework:latest /bin/bash
```

Or, more aggressively, you can build the image and start the container with one command

```
$ sudo docker run -tid [-v /home/<your username>/dsa-share:/root/dsa-share] --
↪name=overgen \
    `sudo docker build . | tail -1 | awk '{ print $3 }'` /bin/bash
```

NOTE: zsh is required. If we use the default bash, the behaviors of our environement setup script are undesired.

1.1.2 Build

Our docker only resolves all the dependences, and clone the repos. Therefore, after the docker container starts, you should build the framework infrastructures from the source code:

```
# Attach to docker container
$ sudo docker attach overgen

# Switch from `bash` to `zsh`, DO NOT use zsh when start docker container
$ zsh

# Inside the docker, enter dsa-framework root folder
$ cd /root/dsa-framework

# Initialize all submodules, SKIP this step if you are using docker
$ ./scripts/init-submodules.sh

# Setup dsa-framework environment variables
$ source ./setup.sh # setup environement variables

# Compile the entire dsa-framework
$ make all -j

# Please source chipyard/env.sh manually if this is a first time build
$ source chipyard/env.sh
```

NOTE: If you just want temporarily leave the container (detach, not close), you should just <Ctrl-p><Ctrl-q> to detach, instead of typing `exit`.

1.1.3 Examples

To verify the repo is successfully built, you can

```
$ cd dsa-apps/demo
$ ./run.sh ss-vecadd.out
```

The command above make a simple vector addition example compiled by LLVM and simulated in Gem5.

All the compiled applications are developed by the same software development kit (SDK), refer to *SDK Section* for more details.

1.1.4 Prebuilt

You can also download a pre-built docker image (~70GB) here, which contains the entire dsa-framework with all toolchains built.

You can import the docker image and use dsa-framework by doing:

```
$ docker import <downloaded tar file>.tar polyarch/dsa-framework:latest
```

1.2 Overview

DSAGEN[1] is a research infrastructure for studying programmable accelerators from the perspective of programming, ISAs, microarchitecture, and scaling.

The principle of this framework is that spatial accelerators can be representend as a graph of simple primitives like network switches, processing elements, memory, and synchronoization. We call this an architecture description graph (ADG). This graph is used not only as a specification of the underlying hardware but also an abstraction to the compiler. The compiler parses the ADG, and map the program (either C+Pragmas or low-level assembly for now) to the hardware. The compiler will take care of figuring out the bitstream format based on the components of the ADG. Finally, optimized kernels are produced as output, composed of control code and the accelerator bitstream. We use a control code to sequence through the accelerator phases, as this reduce the complexity of what is required in most accelerators.

What design space does DSAGEN target?

> Broadly, DSAGEN targets decoupled spatial designs. By "spatial" we mean designs that expose the underlying network to the ISA; depending on the design, other low-level details like operand storage and synchronization of operations are also exposed to the ISA. By "decoupled" we refer to designs which seprate memory pipelines from computation pipelines, so that each can have their own specialized primitives.[2]

TODO(@Jian Weng): Put a figure of mapping program to decoupled-spatial here.

What can DSAGEN be useful for?

- In principle it can be used to study many different aspects of accelerator stacks.
  - At the basic level, DSAGEN can be used as a baseline for other accelerators. DSAGEN is fairly domain-agnostic, so it can be seen as a non-specialized spatial architecture baseline.
  - DSAGEN can also be used in studies of novel spatial architecture features. One can add a new feature, and expose it at the ISA level fairly easily (and with a little more effort add a compiler pass and maybe pragmas to support it).
  - The interactions between CPUs and accelerators are also extremely intersting, including caching, multicore, networks, etc. DSAGEN has an interface with a gem5 core, can be extended to other cores). It currently uses RISCV as the interface.
  - DSAGEN has a spatial architecture compiler that can be independently useful for various other architecture proposals, with hopefully modest implementation overhead.
  - It is our end-goal to be able to deliver reliable hardware generation, although the infrastructure is in the very early stages of being able to supply that.[3]

1.3 DSAGEN Components

1.3.1 Software Stack

In term of functionality, the compilation of decoupled-spatial architecture can be separated into two aspects, host control command, and spatial mapping. We develop a spatial scheduler to map the dataflow graph onto the spatial architecture, and this is available in repo **spatial-scheduler**. Refer [placeholder] for more details on the spatial architecture programming interface and mapping.

[1] DSAGEN can be both used as both Domain-Specific Accelerator Generator and Decoupled Spatial Architecture Generator.

[2] In practice, the lines between memory and computation are blurred, as some memory streams embed computation, and sometimes we are computing an address. The principle remains, however.

[3] We are a small team, and composable hardware design accross the stack takes time. If you are interested in contributing, please let us know!

In term of programming interface, we expose both manually embedded assembly code, and high-level language programming interface. We use RISCV toolchains to extend the ISA encoding for our decoupled-spatial architecture. The ISA encoding patch as well as the macro intrinsics are available in repo **dsa-riscv-ext**. The patch will be applied on **riscv-gnu-toolchain** (as a part of our **chipyard** repo). We use riscv-gnu-toolchain for binary generation.

In order to provide a productive programming interface, we define pragma hints (refer [placeholder] for more detials). We extend the Clang frontend to parse and encode these pragmas, and we implement an LLVM pass to take advantage of this additional information and transform the program into the decoupled-spatial ISA. All these are available in repo **llvm-project**.

1.3.2 Workloads

We have three benchmark suites implemented for demonstration.

MachSuite:
Machsuite is a benchmark suite intended for accelerator-centric research. A subset of workloads are implemented.

DSP:
Digital signal processing (DSP) is a benchmark suite from our prior work REVEL for applications with moderate irregularity and imbalanced execution frequency within loop bodies.

Xilinx Vision:
Xilinx Vitis is a benchmark suite for HLS demonstration. We select a subset of workloads to target.

To develop your own applications, we also provide SDKs for both manual and high-level programming (refer [placeholder] for more details).

1.3.3 Functional Simulation

Our framework extends gem5 by integrating a spatial architecture simulator to simulate the functionality and model the performance of our decoupled-spatial architecture.

1.3.4 RTL Generation and Simulation

@Sihao

1.4 A Simple End-To-End Demo

this is a bunch of filler text. it's not really important. it's just here to make the demo look more realistic. it's not really important. it's just here to make the demo look more realistic. it's not really important. it's just here to make the demo look more realistic.

CHAPTER

TWO

MICRO 2022 TUTORIAL

2.1 Organizers

Sihao Liu, Jian Weng, Dylan Kupsh, Tony Nowatzki
PolyArch Research Group.
University of California, Los Angeles
Date/Time: Sunday October 2nd, 1:00pm - 5:00pm CDT

2.2 Tutorial Overview

As a reaction to the slowing of transistor scaling, significant research has emerged for specialized accelerators, because of their promise of high performance and energy savings. While extremely effective, they require intensive engineering of hardware and software – an effort that must be repeated when new domains arise and when algorithms change.

Ideally, one would be able to generate accelerators based on the behaviors and structure of target applications, and where these applications are specified in a stable and friendly programming interface. In other words, we require the equivalent of high-level synthesis (HLS), but for programmable accelerators – programamble accelerator synthesis. Figure 1 highlights the high-level flow of this paradigm; the compiler simultaneously analyzes multiple kernels, then performs design space exploration using modeling, and ultimately produces optimized kernels along with the accelerator RTL.

The challenges with this paradigm include: How to represent a useful design space, that is broad, easily searchable, and enables significant specialization? How to compile programs from a general language without hindering specialization benefits? How to search this design space efficiently?

In this tutorial, we will present one such approach for programmable accelerator synthesis, along with a corresponding framework: **DSAGEN**, a research infrastructure including compilation, simulation and RTL implementation.

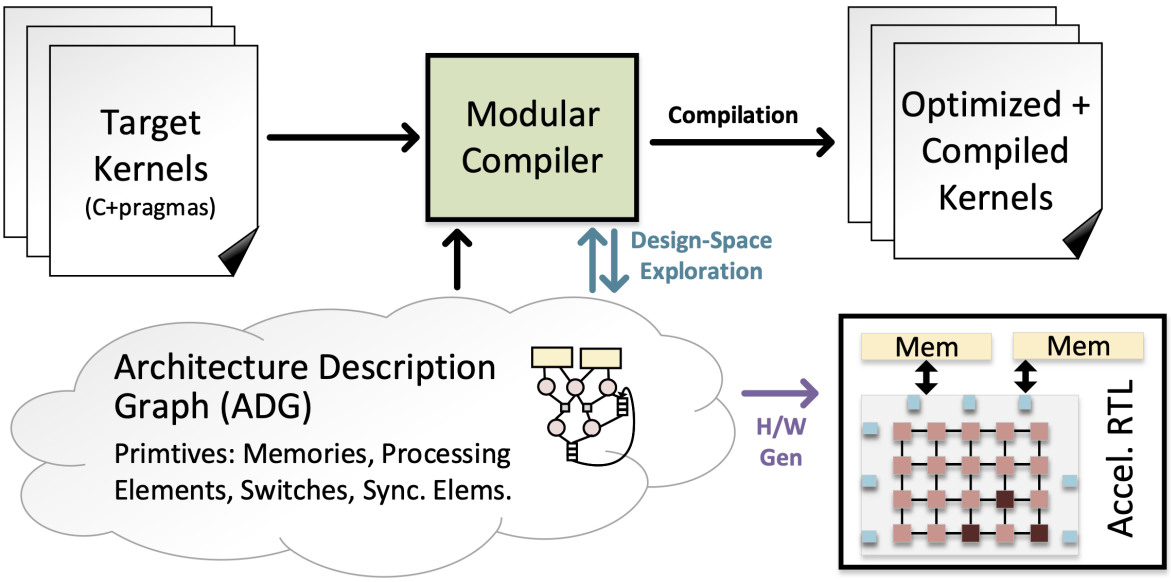


Fig. 1: **Figure 1:** Programmable Accelerator Synthesis.

2.3 Syllabus and Schedule

Introduction (30 Minutes): [slides]

- The Decoupled-Spatial Programming Paradigm
- Composing Hardware with Essential Primatives
- The DSAGEN Framework Stack

Basic Programming (60 Minutes): [slides]

- Introduction to the Automated Compilation flow
  - Annotating Programs with Pragmas
  - Pragma parsing in *clang* and how *llvm* passes interpret to encode data accesses
  - Spatial Mapper Algorithm overview for Decoupled Compuation and Visualization
  - Generating Assembly Code and linking with *gnu-riscv-gcc*
  - Simulating RISC-V binary with gem5
- Hands-on exercises:
  - Change pragma as different compiler transformations
  - Visualize the difference of spatial mapping
  - Simulate RISC-V binary on gem5 simulator to show performance difference

10-Minute Break

Build your own Domain-specific Accelerator (60 Minutes): [slides]

- Introducing the concept of Architecture Description Graph (ADG)

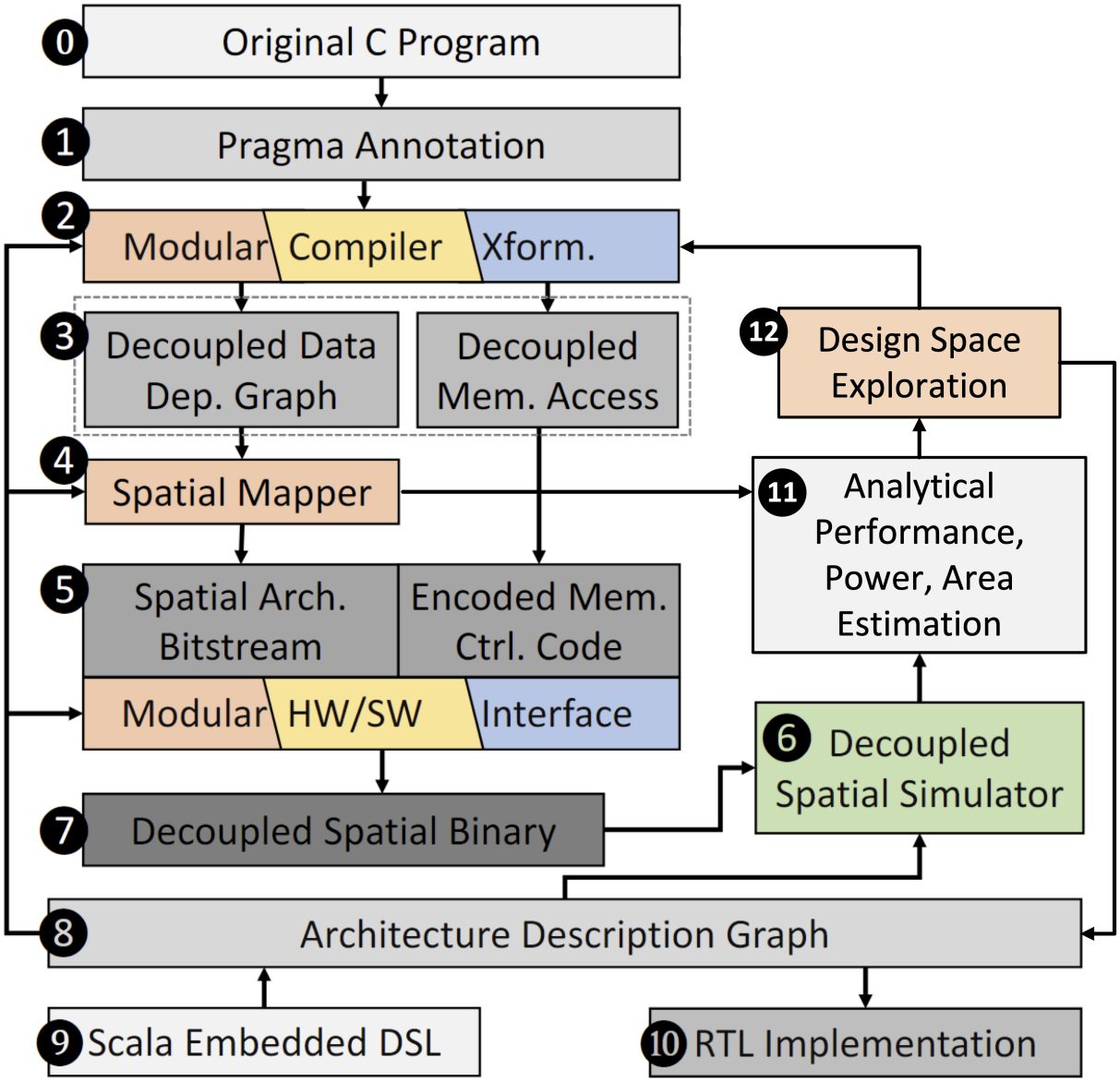


Fig. 2: **Figure 2:** The framework stack of DSAGEN.

  - Design-space and micro-architecture DSA
  - Exporting designs as ADG and simulating RISC-V binary on RTL-level
  - Analytical Power/Performance/Area modeling
- Hands-on exercises:
  - Compose a larger ADG with new operation and different topology by DSL
  - Visualize the difference of ADG and spatial mapping
  - Showing difference of estimated power/performance/area on different hardware designs
  - Performance analysis on how hardware feature affect performance and hardware consumption

10-Minute Break

Automatic Design Space Exploration: (40 Minutes) `[slides]`

- Introduction to the Design Space Explorer
  - An end-to-end flow from a set of programs to auto-generated accelerators
  - Introducing DSE to achieve better Power/Performance/Area
  - Key DSE-specific techniques
- Hands-on exercises:
  - Perform DSE on a small set of kernels
  - Measure performance on generated accelerator
  - Visualize the DSE process and generated accelerator designs

2.4 Installing DSAGEN

To build the DSAGEN Framework, you will need to install Docker. Please follow the instructions on the official docker website.

Start by cloning the repository:

```
$ git clone https://github.com/PolyArch/dsa-framework.git
$ cd dsa-framework
```

Build the docker image:

```
$ docker build .
```

Then, follow this *Setup* page to continue installation of dsa-framework.

2.5 Related Papers

1. Sihao Liu, Jian Weng, Dylan Kupsh, Atefeh Sohrabizadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Lucheng Zhang, Jason Cong and Tong Nowatzki, "OverGen: Improving FPGA Usability through Domain-specific Overlay Generation" in MICRO 2022.
2. Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang and Tony Nowatzki, "DSAGEN: Synthesizing programmable spatial accelerators" in ISCA 2020.
3. Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu and Tong Nowatzki, "A hybrid systolicdataflow architecture for inductive matrix algorithms" in HPCA 2020.
4. Vidushi Dadu, Sihao Liu and Tong Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms" in MICRO 2019.
5. Tong Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karu Sankaralingam, "Stream-Dataflow Acceleration" in ISCA 2017.

CHAPTER

THREE

INSTRUCTION SET ARCHITECTURE

In this section, the semantics of each decoupled-spatial assembly is explained. We will also cover how to hack the GNU infrastructure to add new instructions.

@Sihao?

3.1 Extending RISC-V ISA

This section gives you a quick tour to RISC-V ISA format and slots so that the basic sense and implementation of extending the RISC-V ISA are covered.

These external links are involved, refer them for more details:

- The RISC-V Instruction Set Manual
- risc-v opcode
- RISC-V GNU Compiler Toolchain

3.1.1 Instruction Format

All the RISCV instructions are 32-bit vectors. Please refer page 130 (# on the upper right of each page, not PDF reader page) of the RISC-V manual for the format of these vectors.

To choose a proper format for your extended instruction, the number of src/dst operands, the type of operands (imm/reg), and the number of bits to be encoded should be considered. R, I, and S are recommended — B is just a complicated version of S, and U and J have no *funct3* field, which may occupy the whole line of instruction slots (refer to *next section* for more detials). Therefore, we have 32 bits in total: 7 bits are occupied by the opcode; 3 bits are occupied for *funct3*; each register occupies 5 bits ($2^5=32$ ISA registers); immediate operand can either be 7 or 11 bits.

The instruction format are described in the risc-v opcodes repo, and you can open *opcodes-rv32i*, the most basic module of the RISC-V ISA, for examples. To understand this file, we use addi instruction as an example, and a correspondence to the *ADDI* row in page 130 can be made. Both the figure and the text description are little endian format.

```
addi    rd rs1 imm12           14..12=7 6..2=0x04 1..0=3
```

`rd`, `rs1`, and `imm12` describe the operands of this instruction; `14..12` describes `funct3`; `6..2` describes the opcode. According to Table 24.1 on page 129 (no page number on the PDF), the first two bits are always `11`.

For more information on the operand tokens appear in this file, refer to this for more details. This Python dict declares the bit range this token occupies. The semantics of each token id can be understood by knowing their bit range, acompanied with the figure of the instruction format.

3.1.2 Constraints

To understand the constraints of extending new instructions, we need to know:

1. where are the available slots for the extended instructions, and
2. some additional rules/standards/contraints of each slot.

Refer custom-0/1/2/3 cells in Table 24.1 on page 129. These four slots are reserved for instruction extension.

Refer this file for the operand constraints of each instruction. The operand signature of each instruction should be exactly the same as their corresponding slot in custom.

Do read this!! If you do not want to refactor the ISA aggresively when already having a large project!! You **cannot** give *funct3* random values. The meanings of the 3 bits of *funct3* are critical:

- 1: Send *rs2* from host to the accelerator.
- 2: Send *rs1* from host to the accelerator.
- 4: Receive a value from the accelerator to *rd*.

rs2 only appears when *rs1* appears. Therefore, the bit of *1* cannot be enabled alone. Therefore, *1* and *5* cannot be *funct3*. Also, as mentioned above, if we want to use *0* as *funct3*, we cannot use U-type format.

3.1.3 Assembler Integration

After designing how instructions look like in your mind, we need to integrate them to the compiler, both the **binary encoding** and the **text mnemonic**. This is done by hacking the subrepo, *riscv-gnu-toolchain/riscv-binutils*.

3.1.4 Binary Encoding

To integrate the binary encoding of the extended instruction, we want to replace the code segments 1, 2 related to customized opcodes by the extended encoding.

In risc-v opcodes, scripts are provided to generate these encoding codes. Use the following command:

```
cat opcodes-custom | ./parse-opcode -c > snippet
```

Edit *opcodes-custom* to name the extended instructions, and define the operands.

Not every line of *snippet* is useful, open the file and find the corresponding lines.

Copy those lines and use them to replace the code segments mentioned above.

Mnemonic Format

To integrate the mnemonic (text) format of the extended instruction, we want to add additional rules below this line. The meaning of each column is:

- Name string;
- The default data width; zero means the same as machine bits; here I suggest to give 0;
- The module of the instruction belongs to; here I suggest just give "I", the most basic module;
- The operand description; there is no document for the meaning of each letter, but you can refer to this git issue and read the source code for more details; typically, knowing `s`, `t`, `j`, `d`, and `q` are enough;

- For instructions without aliasing and pesudo representation, the next two columns can just give the `MASK_*` and `MATCH_*` generated in `snippet`.
- I believe it should be something about the aliasing and pseudo thing too, and giving *0* should also suffice.

Implementation

This section includes some our design descisions. Though subjective, we hope this may more or less help your development experience. An auto-patcher is adopted. Refer *dsa-riscv-ext/Makefile* for more details. The path to *riscv-gnu-toolchain* is specified on which the patch is applied. A autopatcher helps:

1. To minimize the invasion to the GNU toolchain and LLVM (so that the cost of rebasing will be minimized when an upstream update is desired);
2. To unify the code hacking interface on both GNU and LLVM;
3. To automate the whole process of code modification by avoiding copy-and-pase, which is error prone.

Refer to isa.ext, I have a text format to describe how the extended instructions look like. Then refer to the Makefile and auto-patch.py for how the involved files are modified to integrate the extended instructions.

CHAPTER

FOUR

PROGRAMMING INTERFACES

This section introduces programming interfaces of our DSAs. We provide both high-level language and embedded assembly code, both in C, programming. Both programming interfaces will be introduced through simple examples.

Because it requires a lengthy process to study the compiler flags and resolve the header file dependencies, we provide software development kits (SDK) for a better developer experience.

4.1 Pragma+C Programming

This section explains both C programming interfaces and our software development kit, which allows users to rapidly start writing your own applications. Please refer to this repo.

4.1.1 Pragmas

To avoid excessive compiler efforts, we adopt a C+pragma programming interface. By simply annotating the program with modest pragmas, the compiler can understand more additional information and encode them in IR metadata.

Here we extend three pragmas:

1. `#pragma ss dfg [unroll(x)]`: This pragma annotates an innermost loop or a compound statement (refer this repo for more details), which indicates the memory accesses and computaiton within the annotated region will be mapped to our decoupled-spatial execution.
   - The `unroll` clause allows users to manually tune the resource occupation of the code region. If `x=-1`, the compiler will automatically explore the unrolling degree.
2. `#pragma ss stream`: This pragma annotates a loop, which indicates all the memory accesses below are restricted. This also indicates the highest loop level to encode memory operations in coarse grain stream commands.
3. `#pragma ss config`: This pragma annotates a compound statement, which indicates all the annotated `dfg` are concurrent on the spatial architecture.

4.1.2 Automated Compilation

In the example vecadd, by simply typing the command below, the generated binaries can be simulated in Gem5.

The explanation is separated into two aspects, the programming interfaces, and the build infrastructures. To explain the programming interfaces, we provide a set unified interfaces (in this case, declared in `common/interface.h` and implemented in `vecadd.c`) for you to write application kernels and model its performance.

1. `struct Arguments` are the input of the benchmark kernel, which will be initialized by `init_data` and used as input argument of `run_*`.
2. `init_data` initializes the input of application. We provide several convinence function macros in `common/test.h` to initialize the data.
3. `run_reference` is the function invoke the host execution for a golden reference of the application result.
4. `run_accelerator` is the function to invoke the accelerator. The `is_warmup` indicates if it is cache warmup invocation.
5. `sanity_check` verifies the result of compilation. We provide several convinence function macros in `common/test.h` to check the result correctness.

Feel free to copy and rename `vecadd.c` and write other kernels and use the following command to simulate.

`%` is the name of the the kernel c file without suffix. All the files share the same main function implemented in `common/gem5-harness.c` — the main function invokes each function sequentially, and invokes `run_accelerator` twice to warm up the cache and time it.

To explain the build infrastructures, we overview the flow of compilation:

1. The kernel file is first parsed by our extended `clang` and generate an LLVM IR file (see `vecadd.ll`).
2. This IR file is fed to an LLVM pass for decoupled-spatial transformation.
   - The decoupled memory access are encoded in control commands and embedded in the host assemly code (see `ss-vecadd.ll`).
   - The decoupled computation are in dfg file(s) (see `vecadd_%.dfg` where % is the unrolling degree).
3. The transformed IR is fed to LLVM code generator to generate assembly code (see `ss-vecadd.s`).
4. The generated assembly code will be fed to riscv-gnu linker to generate the binaries (see `ss-vecadd.out`).

Because Chipyard Rocket core adopts a different model of RISCV CPU as Gem5 implements, it requires different compilation flags and link options. For the RTL simulation purpose, by simply type

The Chipyard compatible main function will be linked.

4.2 Embedded ASM and DFG

this is a bunch of filler text. it's not really important. it's just here to make the demo look more realistic. it's not really important. it's just here to make the demo look more realistic. it's not really important. it's just here to make the demo look more realistic.

CHAPTER

FIVE

DATAFLOW GRAPH

The Dataflow Graph (DFG) is a representation of the dataflow of a program. It is a directed graph where the nodes are the operations and the edges are the data dependencies between the operations. The DFG is a static representation of the dataflow of a program. It is not a representation of the actual dataflow at runtime.

The Compiler automatically creates DFG files. The DFG files are used by the simulator and scheduler to map onto the actual hardware. DFG files can also be manually created, this section acts as a reference on reading and creating custom DFG files.

5.1 DFG File Format

The dfgfile contains 4 parts:

1. Array Declaration
2. Port Declaration
3. Operation Declaration
4. Meta-level information

We will go through each of these sections seperately.

5.1.1 Array Declaration

Arrays can be declared with the following Format

[array-type] <array-name> <size>

where the array-type can be one of the following: * *dma* - Direct Memory Access * *spm* - Scratchpad * *rec* - Recurrance * *gen* - Generate * *reg* - Register

5.1.2 Port Declaration

Inputs can be declared with the following format:

Input[Size] <input-name>[<vectorization-degree>] source=<array-name> [stated]

Correspondingly, Outputs can be declared with the following format:

Output[Size] <output-name>[<vectorization-degree>] destination=<array-name> [stated]

Size refers to the datatype size. For instance, Input64 would assume a 64-bit stream while a Input32 would assume a 32-bit stream. If no size is specified, the scheduler will default to creating a 64-bit stream.

Note: The scheduler currently supports decomposible routing. Thus, it can combine different datatypes. However, this is not currently supported by the hardware generator and simulator. Thus, it is recommended to use the same datatype for all streams. We plan to fix this in an upcoming release.

The vectorization degree refers to the number of elements in the stream. For instance, Input64[2] would assume a 64-bit stream with 2 elements. If no vectorization degree is specified, the scheduler will default to creating a 1-element stream.

The source and destination fields refer to the array that the input and output are mapped to. An edge will be created from the specified array to the input/output port.

Stated refers to the first element of the stream being used as a control element within a Operation. By default, ports are not stated and if the 'stated' keyword is specified, the port will be stated.

Routing Ports

Port variable names will automatically be created under the format:

<input/output-name>_<element-number>

Thus, if we declared a port with the name *foo* and the vectorization degree of 2, the data elements would be named *foo_0* and *foo_1*.

Additionally, the stated element (if the port is stated) will be created under the format:

<input/output-name>_State

These variables can be directly used within future operations. Additionally elements can be renamed with the following format:

<new_name> = <old-name>

The compiler uses this naming feature to rename the final operation results to the name of the output port.

Optional Port Reuse Pragmas

The compiler automatically generates pragmas describing memory stream reuse information. These pragmas are optional; they are not used in scheduling the DFG to the ADG and only used by the DSE performance models.

The compiler generates the following pragmas for both input and output ports:

#pragma cmd <cmd-coefficient> #pragma repeat <repeat-rate> #pragma reuse <reuse-rate>

The cmd coefficient refers to a bound on the memory traffic for a stream, due to a command required to load the data. For instance, in a indirect access stream where the address must be generated by scalar operations, the cmd increases. By default cmd is 1.

The repeat rate refers to the number of times a stream is repeated. For instance, if a stream is used in a loop, the repeat rate is the number of times the loop is executed. By default, the repeat rate is 1.

The reuse rate refers to the number of times a stream is reused by the L2 cache. In the compiler, this is generated by a reuse analysis pass. By default, the reuse rate is 0.

Operation Declaration and Mapping

Operations can be declared with the following format:

<operation-result> = <operation-name>(<operation-arguments>)

where the operation-name can be any operation described within the ISA. The operation-arguments are the inputs to the operation. Each operation argument should be seperated by a comma. The operation-result is the output of the operation.

Stated Operation

Operations that utilize the stated control argument have the additional parameter as follows:

ctrl=$<Port-Name>_State & 8{0: d, 8: r}

This declares that the result will depend upon the first 8 bits of the stated link.

5.1.3 Meta-level Information

Each DFG-file can have multiple subgraphs. Each subgraph is seperated by:

—-

The compiler always produces the first sub-dfg as the array-declaration. Variables within different sub-dfgs should be named seperately and the Arrays are the only variable that can be used across multiple sub-dfgs

Subdfgs have the following optional pragmas:

#pragma group frequency <code-execution-frequency> #pragma group unroll <vectorization-degree>

The frequency pragma refers to the code-execution frequency of the sub-dfg. This code frequency will be used by the DSE performance models to determine relative execution time for each sub-dfg. By default, the frequency is 1.

The unroll pragma refers to the vectorization degree of the sub-dfg. This pragma is currently only used when determing recurrance bottleneck. By default, the unroll degree is 1.

The DFG parser also supports comments with lines that are preceded by a hashtag (#). The last line of the dfgfile must also be a blank space.

5.2 DFG File Examples

5.2.1 Accumulate Example

The following is an example of a DFG file for a non-vectorized add operation:

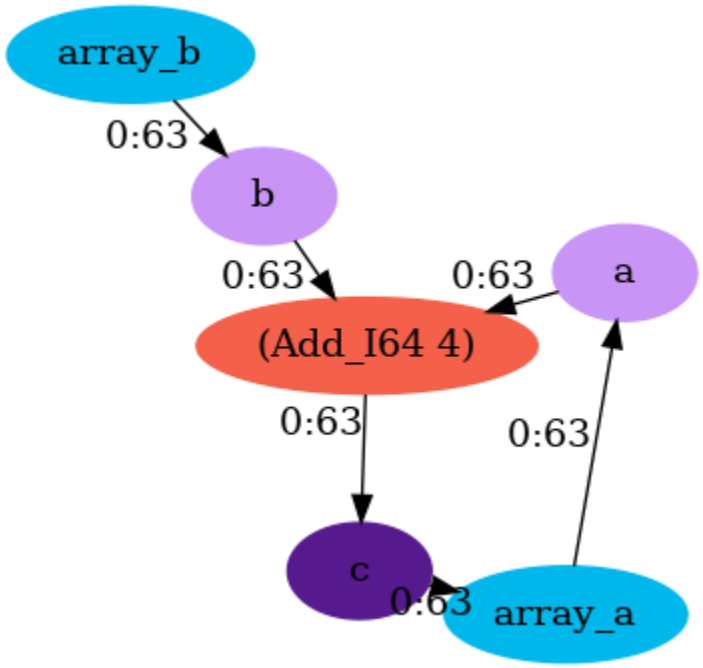
```
# Declare sub-dfg meta properties
# Frequency is 0 as no work happens in this sub-dfg
#pragma group frequency 0

# Array Declaration
Array: array_a 131072 dma
Array: array_b 131072 dma

----
```

(continues on next page)

(continued from previous page)

```
# Declare sub-dfg meta properties

#pragma group frequency 255
#pragma group unroll 1

# Port Declaration
Input64: a source=array_a
Input64: b source=array_b

# Operation Declaration
c = Add_I64(a, b)

# Output Declaration
Output64: c destination=array_a
```

This produces a dataflow graph that looks like the following:



5.2.2 Acc Vectorization Example

The following is an example of a DFG file for a vectorized-by-four add operation:

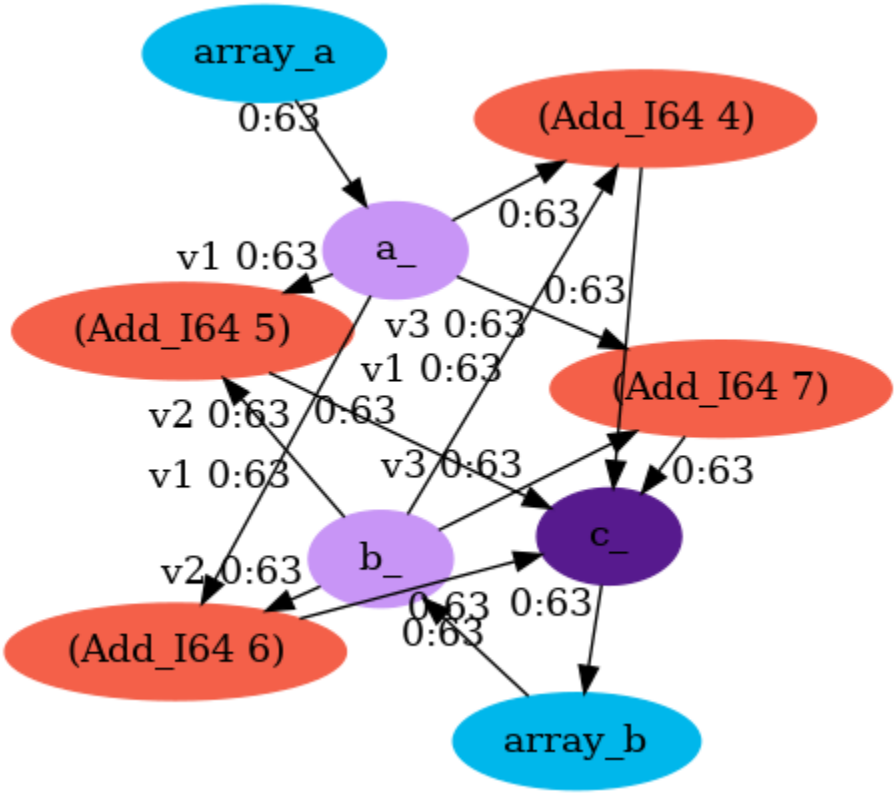
```
# Declare sub-dfg meta properties
# Frequency is 0 as no work happens in this sub-dfg
#pragma group frequency 0

# Array Declaration
Array: array_a 131072 dma
Array: array_b 131072 dma
```

(continues on next page)

(continued from previous page)

```
----
# Declare sub-dfg meta properties

#pragma group frequency 255
#pragma group unroll 4

# Port Declaration
Input64: a_[4] source=array_a
Input64: b_[4] source=array_b

# Operation Declaration
c_0 = Add_I64(a_0, b_1)
c_1 = Add_I64(a_1, b_1)
c_2 = Add_I64(a_2, b_2)
c_3 = Add_I64(a_3, b_3)

# Output Declaration
Output64: c_[4] destination=array_b
```

This produces a dataflow graph that looks like the following:



5.2.3 Complex Example

This is an example of a manually programmed DFG for the Stencil-2d workload.

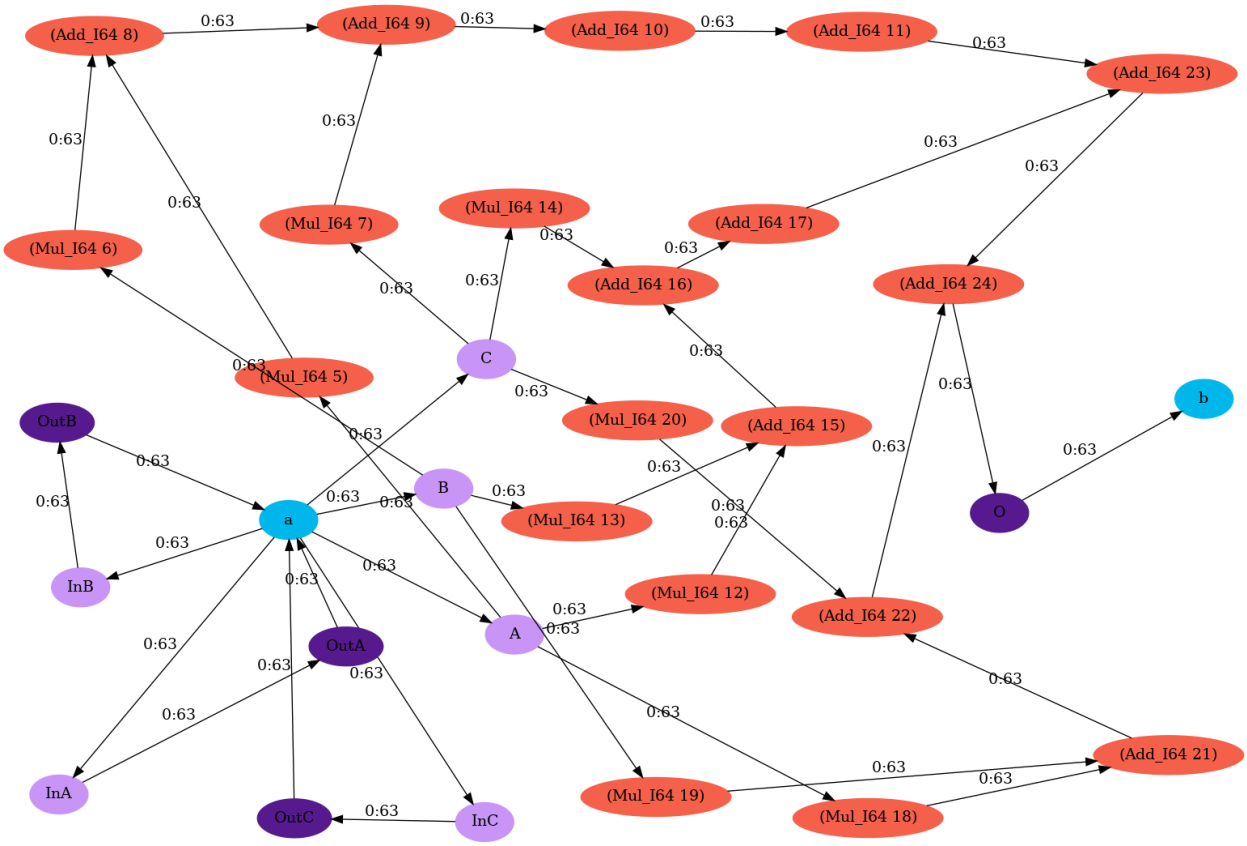
```
# Declare sub-dfg meta properties
# Frequency is 0 as no work happens in this sub-dfg
#pragma group frequency 0


# Array Declaration
Array: a 9248 dma
Array: b 8192 dma

----
# Declare sub-dfg meta properties
# Most of the work happens here so we can set the frequency to 90 or 90%
#pragma group frequency 90
#pragma group unroll 1

# Declare the input ports

#pragma reuse=0.66
Input64: A source=a
#pragma reuse=0.66
Input64: B source=a
#pragma reuse=0.66
Input64: C source=a

# Do the operations
MUL_0A = Mul_I64(A, $Reg0)
MUL_0B = Mul_I64(B, $Reg0)
MUL_0C = Mul_I64(C, $Reg0)

TMPS0 = Add_I64(MUL_0A, MUL_0B)
PSUM0 = Add_I64(MUL_0C, TMPS0)

SHIFT0_REG0 = Add_I64(PSUM0, $Reg0)
SHIFT0_REG1 = Add_I64(SHIFT0_REG0, $Reg0)

MUL_1A = Mul_I64(A, $Reg0)
MUL_1B = Mul_I64(B, $Reg0)
MUL_1C = Mul_I64(C, $Reg0)

TMPS1 = Add_I64(MUL_1A, MUL_1B)
PSUM1 = Add_I64(MUL_1C, TMPS1)

SHIFT1_REG0 = Add_I64(PSUM1, $Reg0)

MUL_2A = Mul_I64(A, $Reg0)
MUL_2B = Mul_I64(B, $Reg0)
MUL_2C = Mul_I64(C, $Reg0)

TMPS2 = Add_I64(MUL_2A, MUL_2B)
```

(continues on next page)

(continued from previous page)

```
PSUM2 = Add_I64(MUL_2C, TMPS2)

PSUM3 = Add_I64(SHIFT0_REG1, SHIFT1_REG0)
O = Add_I64(PSUM3, PSUM2)

# Declare the output ports (there is no reuse)
Output64: O destination=b

----
# Declare sub-dfg meta properties
#pragma group frequency 3

# These are indirect stream generators
Input64: InA source=a
OutA = InA
Output64: OutA destination=a

----
# Declare sub-dfg meta properties
#pragma group frequency 3

# These are indirect stream generators

Input64: InB source=a
OutB = InB
Output64: OutB destination=a

----
# Declare sub-dfg meta properties
#pragma group frequency 3

# These are indirect stream generators

Input64: InC source=a
OutC = InC
Output64: OutC destination=a
```

The resulting dataflow graph looks like the following:


(Add_I64 8)
(Add_I64 9)
(Add_I64 10)
(Add_I64 11)
(Add_I64 23)
(Mul_I64 6)
(Mul_I64 7)
(Mul_I64 14)
(Add_I64 17)
(Add_I64 16)
(Add_I64 24)
(Mul_I64 5)
C
(Mul_I64 20)
(Add_I64 15)
b
OutB
B
(Mul_I64 13)
O
a
InB
(Mul_I64 12)
(Add_I64 22)
A
OutA
(Add_I64 21)
InA
(Mul_I64 19)
(Mul_I64 18)
OutC
InC
0:63

CHAPTER

SIX

ARCHITECTURE DESCRIPTION GRAPH

The Architecture Description Graph describes the underlying CGRA hardware capabilities. It is a directive graph between different hardware nodes.

The ADG is generated by both the chipyard generator and as a result of the Design Space Explorer (DSE). The ADG also serves as an input into both the DSE and Chipyard generator, to create better hardware designs and simulate capabilites. The scheduler schedules DFG graphs onto the ADG.

6.1 ADG File Format

ADG Files contain two parts:

- ADG Module Declaration
- ADG Link Declaration

6.1.1 ADG Module Declaration

The ADG is composed of different hardware modules, or nodes, with each containing their own attributes. Broadly, the ADG demarcates three different node types (*spatial*, *sync*, and *data* nodes) based on their typical placement and function within the adg.

Spatial Nodes

Spatial Nodes perform the computation and routing network inside the ADG. Consisting of processing elements and switches, these nodes are interlinked, performing computation and recieving inputs/outputs from the sync nodes.

Processing Elements

Processing Elements are the basic computational unit of the ADG. They are the nodes that perform the actual computation. Each processing element has a set of defined operations, taking inputs and then performing the operation upon it.

Passthroughs

Processing Elements can act as passthroughs, or perform the copy operation, during scheduling. This is useful to allow generality in scheduling.

Switches

Switches perform routing within the ADG, allowing greater generality in designs. In hardware, switches act as a series of muxes allowing data to move from any input to any output.

Broadcast

Switches are also helpful as they have the capability to broadcast, or one input go to two different outputs. This functionality is required for several schedules where broadcasting is needed. Processing elements are not able to broadcast data.

Spatial Node Properties

Fifo Depths

Each spatial node has a fifo, allowing it to balance delays and hopefully remove pipeline stalls. These fifos can be set by the *fifo_depth* property. Currently, the fifo can't be eliminated without potentially hurting the frequency, thus the fifo depth must be set to at least 1.

Sync Nodes

Sync Nodes bring data into the spatial architecture from the data. It consists of input vector ports and output vector ports.

Input Vector Ports

Input vector ports act as the input. They generate the data requests and stream data into the spatial part of the ADG.

Output Vector Ports

Output vector ports act as the output. These hardware modules take data produced from the spatial architecture and then feed them into different data nodes.

Sync Node Properties

Stated

Both Input and Output Vector ports can be stated, meaning the first link is reserved for the stated control inputs from the DFG.

Data Nodes

Data nodes interact with memory, and deal with streaming requests and different levels of the memory heirarchy. Currently, there are 5 different data node types, *DMA*, *Scratchpad*, *Recurrance*, *Generate*, and *Register*. Each node performs different types of data movement, and has its own associated functionality.

DMA

DMA nodes stream data from the DRAM and L2 cache.

Scratchpad

Scratchpad nodes act as a private cache for each accelerator tile. The scratchpad has an associated size and is replicated within each tile.

Recurrance

Recurrance Nodes directly stream data from the output back into the input vector port.

Generate

Register

Data Node Properties

Data Nodes all interconnect on a bus. Thus, the bandwidth mechanism works similarly for all data nodes, depending on their replication across cores.

6.2 ADG Visualization

We have developed two different methods (one using graphviz and another using html) to visualize the adg, each having their own tradeoffs. Both these methods are useful in gaining an underlying insight into ADG structure.

6.2.1 Using Dot Files

To get an ADG Graphviz file, you must first run the scheduler using the following command:

```
ss_sched adg.json -f
```

A graphviz file should appear in the newly created viz directory. To view the dot file, you must first install the graphviz package. Then, you can run the following command:

```
dot -Tpng viz/adg.gv -o viz/adg.png
```

Alternatively, we have found more structured results using:

```
neato -Goverlap=false -Gstart=self -Gepsilon=.0000001 -Tpng -o viz/adg.png viz/adg.gv
```

6.2.2 Visualizing Using HTML

To get a HTML visualization of the ADG, you must run the python script *adg_visualize.py* on the file specified file. Thus, it looks like this:

```
python3 adg_visualize.py adg.json
```

Then, you can open the generated html file in your browser to view the ADG. This script is interactive, allowing a rearrangement of modules. We have also found the physics-based simulation to be more instructive, producing a grid-like format for mesh designs, which hasn't necessarily been true of graphviz-based designs.

6.3 ADG File Example

The following is an example of an ADG File that is a 2 x 2 Mesh, with 2 input ports and 2 output ports. The processing elements all only have one operation, and the data engines are connected to every port:

```
{
"DSAGenNodes" : {
    "ProcessingElement.0" : {
    "ConfigBitEncode" : {
        "Enabled" : [ 0, 0 ],
        "Instruction_0_Valid" : [ 1, 1 ],
        "Instruction_0_OperandSel_0" : [ 4, 2 ],
        "Instruction_0_OperandSel_1" : [ 7, 5 ],
        "Instruction_0_CtrlMode" : [ 9, 8 ],
        "Instruction_0_CtrlInputSel" : [ 12, 10 ],
        "Instruction_0_Opcode" : [ 16, 13 ],
        "Instruction_0_ResultOut_0" : [ 17, 17 ],
        "Instruction_0_ResultReg_0" : [ 18, 18 ],
        "Instruction_0_Latency" : [ 21, 19 ],
        "MetaCtrlEntry_0_valid" : [ 22, 22 ],
        "MetaCtrlEntry_0_reuseOperand" : [ 24, 23 ],
        "MetaCtrlEntry_0_discardResult" : [ 25, 25 ],
        "MetaCtrlEntry_0_resetReg" : [ 26, 26 ],
        "MetaCtrlEntry_0_abstain" : [ 27, 27 ],
        "MetaCtrlEntry_1_valid" : [ 28, 28 ],
        "MetaCtrlEntry_1_reuseOperand" : [ 30, 29 ],
        "MetaCtrlEntry_1_discardResult" : [ 31, 31 ],
        "MetaCtrlEntry_1_resetReg" : [ 32, 32 ],
        "MetaCtrlEntry_1_abstain" : [ 33, 33 ],
        "MetaCtrlEntry_2_valid" : [ 34, 34 ],
        "MetaCtrlEntry_2_reuseOperand" : [ 36, 35 ],
        "MetaCtrlEntry_2_discardResult" : [ 37, 37 ],
        "MetaCtrlEntry_2_resetReg" : [ 38, 38 ],
        "MetaCtrlEntry_2_abstain" : [ 39, 39 ],
        "MetaCtrlEntry_3_valid" : [ 40, 40 ],
        "MetaCtrlEntry_3_reuseOperand" : [ 42, 41 ],
        "MetaCtrlEntry_3_discardResult" : [ 43, 43 ],
        "MetaCtrlEntry_3_resetReg" : [ 44, 44 ],
        "MetaCtrlEntry_3_abstain" : [ 45, 45 ]
    },
    "dsagen2.comp.config.CompKeys$CompNode$" : {
```

(continues on next page)

(continued from previous page)

```
        "compBits" : 64,
        "comment" : "row0_col0",
        "parameterClassName" : "dsagen2.comp.config.CompNodeParameters",
        "compUnitBits" : 64,
        "nodeType" : "ProcessingElement",
        "nodeId" : 0,
        "supportNodeActive" : true
    },
    "dsagen2.comp.config.CompKeys$OutputBuffer$" : {
        "outputBufferDepth" : 4,
        "parameterClassName" : "dsagen2.comp.config.common.CompNodeOutputBufferParameters
↪",
        "staticOutputBuffer" : false
    },
    "dsagen2.comp.config.CompKeys$RegFile$" : {
        "numReg" : 1,
        "asyncRF" : true,
        "update" : true,
        "parameterClassName" : "dsagen2.comp.config.processing_element.
↪PERegFileParameters",
        "resetRegIdx" : [ 0 ]
    },
    "dsagen2.comp.config.CompKeys$MetaControl$" : {
        "outputLSBCtrl" : true,
        "sizeLUT" : 4,
        "abstain" : true,
        "parameterClassName" : "dsagen2.comp.config.processing_element.
↪PEMetaCtrlParameters",
        "inputLSBCtrl" : true,
        "reuseOperand" : true,
        "resetRegister" : true,
        "discardResult" : true
    },
    "dsagen2.comp.config.CompKeys$DsaOperations$" : {
        "isDynamic" : true,
        "OperationDataTypeSet" : [ "Copy", "Add_I64", "FAdd_D64", "FMul_D64", "FSub_D64",
↪ "Max_I64", "Min_I64", "Mul_I64", "Sub_I64" ],
        "maxInstRepeatTime" : 0,
        "definedLatency" : 0,
        "parameterClassName" : "dsagen2.comp.config.processing_element.
↪PEDsaOperationParameters",
        "instSlotSize" : 1,
        "maxFifoDepth" : 4
    }
    },
    "Switch.3" : {
    "ConfigBitEncode" : {
        "Enabled" : [ 0, 0 ],
        "SwitchRouting$_0_SubNet_0" : [ 3, 1 ],
        "SwitchRouting$_1_SubNet_0" : [ 6, 4 ],
        "SwitchRouting$_2_SubNet_0" : [ 9, 7 ],
        "SwitchRouting$_3_SubNet_0" : [ 12, 10 ]
```

(continues on next page)

(continued from previous page)


```
    },
    "dsagen2.comp.config.CompKeys$CompNode$" : {
        "compBits" : 64,
        "comment" : "row1_col1",
        "parameterClassName" : "dsagen2.comp.config.CompNodeParameters",
        "compUnitBits" : 64,
        "nodeType" : "Switch",
        "nodeId" : 3,
        "supportNodeActive" : true
    },
    "dsagen2.comp.config.CompKeys$OutputBuffer$" : {
        "outputBufferDepth" : 4,
        "parameterClassName" : "dsagen2.comp.config.common.CompNodeOutputBufferParameters
↪",
        "staticOutputBuffer" : false
    },
    "dsagen2.comp.config.CompKeys$SwitchRouting$" : {
        "initFullMatrix" : [ ],
        "parameterClassName" : "dsagen2.comp.config.switch.SWRoutingParameters",
        "initIndividualMatrix" : [ ]
    }
    },
    "Switch.2" : {
    "ConfigBitEncode" : {
        "Enabled" : [ 0, 0 ],
        "SwitchRouting$_0_SubNet_0" : [ 2, 1 ],
        "SwitchRouting$_1_SubNet_0" : [ 4, 3 ],
        "SwitchRouting$_2_SubNet_0" : [ 6, 5 ],
        "SwitchRouting$_3_SubNet_0" : [ 8, 7 ]
    },
    "dsagen2.comp.config.CompKeys$CompNode$" : {
        "compBits" : 64,
        "comment" : "row1_col0",
        "parameterClassName" : "dsagen2.comp.config.CompNodeParameters",
        "compUnitBits" : 64,
        "nodeType" : "Switch",
        "nodeId" : 2,
        "supportNodeActive" : true
    },
    "dsagen2.comp.config.CompKeys$OutputBuffer$" : {
        "outputBufferDepth" : 4,
        "parameterClassName" : "dsagen2.comp.config.common.CompNodeOutputBufferParameters
↪",
        "staticOutputBuffer" : false
    },
    "dsagen2.comp.config.CompKeys$SwitchRouting$" : {
        "initFullMatrix" : [ ],
        "parameterClassName" : "dsagen2.comp.config.switch.SWRoutingParameters",
        "initIndividualMatrix" : [ ]
    }
    },
    "Switch.1" : {
```


(continues on next page)

(continued from previous page)

```
    "ConfigBitEncode" : {
        "Enabled" : [ 0, 0 ],
        "SwitchRouting$_0_SubNet_0" : [ 2, 1 ],
        "SwitchRouting$_1_SubNet_0" : [ 4, 3 ],
        "SwitchRouting$_2_SubNet_0" : [ 6, 5 ],
        "SwitchRouting$_3_SubNet_0" : [ 8, 7 ]
    },
    "dsagen2.comp.config.CompKeys$CompNode$" : {
        "compBits" : 64,
        "comment" : "row0_col1",
        "parameterClassName" : "dsagen2.comp.config.CompNodeParameters",
        "compUnitBits" : 64,
        "nodeType" : "Switch",
        "nodeId" : 1,
        "supportNodeActive" : true
    },
    "dsagen2.comp.config.CompKeys$OutputBuffer$" : {
        "outputBufferDepth" : 4,
        "parameterClassName" : "dsagen2.comp.config.common.CompNodeOutputBufferParameters
↪",
        "staticOutputBuffer" : false
    },
    "dsagen2.comp.config.CompKeys$SwitchRouting$" : {
        "initFullMatrix" : [ ],
        "parameterClassName" : "dsagen2.comp.config.switch.SWRoutingParameters",
        "initIndividualMatrix" : [ ]
    }
    },
    "Switch.0" : {
    "ConfigBitEncode" : {
        "Enabled" : [ 0, 0 ],
        "SwitchRouting$_0_SubNet_0" : [ 2, 1 ],
        "SwitchRouting$_1_SubNet_0" : [ 4, 3 ],
        "SwitchRouting$_2_SubNet_0" : [ 6, 5 ],
        "SwitchRouting$_3_SubNet_0" : [ 8, 7 ]
    },
    "dsagen2.comp.config.CompKeys$CompNode$" : {
        "compBits" : 64,
        "comment" : "row0_col0",
        "parameterClassName" : "dsagen2.comp.config.CompNodeParameters",
        "compUnitBits" : 64,
        "nodeType" : "Switch",
        "nodeId" : 0,
        "supportNodeActive" : true
    },
    "dsagen2.comp.config.CompKeys$OutputBuffer$" : {
        "outputBufferDepth" : 4,
        "parameterClassName" : "dsagen2.comp.config.common.CompNodeOutputBufferParameters
↪",
        "staticOutputBuffer" : false
    },
    "dsagen2.comp.config.CompKeys$SwitchRouting$" : {
```

(continues on next page)

(continued from previous page)


```
        "initFullMatrix" : [ ],
        "parameterClassName" : "dsagen2.comp.config.switch.SWRoutingParameters",
        "initIndividualMatrix" : [ ]
    }
    },
    "RecurrenceEngine.0" : {
    "dsagen2.mem.config.MemKeys$MemNode$" : {
        "numWrite" : 1,
        "memUnitBits" : 8,
        "numRead" : 1,
        "MaxLength1D" : 2147483646,
        "parameterClassName" : "dsagen2.mem.config.MemNodeParameters",
        "MaxLength3D" : 0,
        "capacity" : 16384,
        "LinearLength1DStream" : false,
        "numGenDataType" : 0,
        "LinearPadding" : true,
        "MaxAbsStretch3D2D" : 0,
        "NumLength1DUnitBitsExp" : 0,
        "MaxAbsStride3D" : 0,
        "MaxAbsStride1D" : 1,
        "IndirectStride2DStream" : false,
        "AtomicOperations" : [ ],
        "NumIdxUnitBitsExp" : 0,
        "MaxAbsDeltaStride2D" : 0,
        "LinearStride2DStream" : false,
        "MaxLength2D" : 0,
        "MaxAbsStretch2D" : 0,
        "nodeType" : "RecurrenceEngine",
        "supportBuffet" : false,
        "numMemUnitBitsExp" : 4,
        "MaxAbsStretch3D1D" : 0,
        "IndirectIndexStream" : false,
        "NumStride2DUnitBitsExp" : 0,
        "writeWidth" : 32,
        "MaxAbsStride2D" : 0,
        "numPendingRequest" : 0,
        "readWidth" : 32,
        "nodeId" : 0,
        "streamStated" : true,
        "numSpmBank" : 0,
        "IndirectLength1DStream" : false,
        "MaxAbsDeltaStretch2D" : 0
    }
    },
    "RegisterEngine.0" : {
    "dsagen2.mem.config.MemKeys$MemNode$" : {
        "numWrite" : 1,
        "memUnitBits" : 8,
        "numRead" : 1,
        "MaxLength1D" : 0,
        "parameterClassName" : "dsagen2.mem.config.MemNodeParameters",
```


(continues on next page)

(continued from previous page)


```
        "MaxLength3D" : 0,
        "capacity" : 16384,
        "LinearLength1DStream" : false,
        "numGenDataType" : 0,
        "LinearPadding" : false,
        "MaxAbsStretch3D2D" : 0,
        "NumLength1DUnitBitsExp" : 0,
        "MaxAbsStride3D" : 0,
        "MaxAbsStride1D" : 0,
        "IndirectStride2DStream" : false,
        "AtomicOperations" : [ ],
        "NumIdxUnitBitsExp" : 0,
        "MaxAbsDeltaStride2D" : 0,
        "LinearStride2DStream" : false,
        "MaxLength2D" : 0,
        "MaxAbsStretch2D" : 0,
        "nodeType" : "RegisterEngine",
        "supportBuffet" : false,
        "numMemUnitBitsExp" : 4,
        "MaxAbsStretch3D1D" : 0,
        "IndirectIndexStream" : false,
        "NumStride2DUnitBitsExp" : 0,
        "writeWidth" : 8,
        "MaxAbsStride2D" : 0,
        "numPendingRequest" : 0,
        "readWidth" : 8,
        "nodeId" : 0,
        "streamStated" : true,
        "numSpmBank" : 0,
        "IndirectLength1DStream" : false,
        "MaxAbsDeltaStretch2D" : 0
    }
    },
    "GenerateEngine.0" : {
    "dsagen2.mem.config.MemKeys$MemNode$" : {
        "numWrite" : 0,
        "memUnitBits" : 8,
        "numRead" : 1,
        "MaxLength1D" : 2147483646,
        "parameterClassName" : "dsagen2.mem.config.MemNodeParameters",
        "MaxLength3D" : 0,
        "capacity" : 16384,
        "LinearLength1DStream" : true,
        "numGenDataType" : 4,
        "LinearPadding" : true,
        "MaxAbsStretch3D2D" : 0,
        "NumLength1DUnitBitsExp" : 4,
        "MaxAbsStride3D" : 0,
        "MaxAbsStride1D" : 1,
        "IndirectStride2DStream" : true,
        "AtomicOperations" : [ ],
        "NumIdxUnitBitsExp" : 4,
```


(continues on next page)

(continued from previous page)


```
        "MaxAbsDeltaStride2D" : 0,
        "LinearStride2DStream" : true,
        "MaxLength2D" : 2147483646,
        "MaxAbsStretch2D" : 1073741822,
        "nodeType" : "GenerateEngine",
        "supportBuffet" : false,
        "numMemUnitBitsExp" : 4,
        "MaxAbsStretch3D1D" : 0,
        "IndirectIndexStream" : true,
        "NumStride2DUnitBitsExp" : 4,
        "writeWidth" : 0,
        "MaxAbsStride2D" : 1073741822,
        "numPendingRequest" : 16,
        "readWidth" : 8,
        "nodeId" : 0,
        "streamStated" : true,
        "numSpmBank" : 0,
        "IndirectLength1DStream" : true,
        "MaxAbsDeltaStretch2D" : 0
    }
    },
    "ScratchpadMemory.0" : {
    "dsagen2.mem.config.MemKeys$MemNode$" : {
        "numWrite" : 1,
        "memUnitBits" : 8,
        "numRead" : 1,
        "MaxLength1D" : 2147483646,
        "parameterClassName" : "dsagen2.mem.config.MemNodeParameters",
        "MaxLength3D" : 0,
        "capacity" : 524288,
        "LinearLength1DStream" : true,
        "numGenDataType" : 0,
        "LinearPadding" : true,
        "MaxAbsStretch3D2D" : 0,
        "NumLength1DUnitBitsExp" : 4,
        "MaxAbsStride3D" : 0,
        "MaxAbsStride1D" : 1,
        "IndirectStride2DStream" : true,
        "AtomicOperations" : [ ],
        "NumIdxUnitBitsExp" : 4,
        "MaxAbsDeltaStride2D" : 0,
        "LinearStride2DStream" : true,
        "MaxLength2D" : 2147483646,
        "MaxAbsStretch2D" : 1073741822,
        "nodeType" : "ScratchpadMemory",
        "supportBuffet" : false,
        "numMemUnitBitsExp" : 4,
        "MaxAbsStretch3D1D" : 0,
        "IndirectIndexStream" : true,
        "NumStride2DUnitBitsExp" : 4,
        "writeWidth" : 32,
        "MaxAbsStride2D" : 1073741822,
```


(continues on next page)

(continued from previous page)


```
        "numPendingRequest" : 16,
        "readWidth" : 32,
        "nodeId" : 0,
        "streamStated" : true,
        "numSpmBank" : 4,
        "IndirectLength1DStream" : true,
        "MaxAbsDeltaStretch2D" : 0
    }
    },
    "DirectMemoryAccess.0" : {
    "dsagen2.mem.config.MemKeys$MemNode$" : {
        "numWrite" : 1,
        "memUnitBits" : 8,
        "numRead" : 1,
        "MaxLength1D" : 2147483646,
        "parameterClassName" : "dsagen2.mem.config.MemNodeParameters",
        "MaxLength3D" : 0,
        "capacity" : 1099511627776,
        "LinearLength1DStream" : true,
        "numGenDataType" : 0,
        "LinearPadding" : true,
        "MaxAbsStretch3D2D" : 0,
        "NumLength1DUnitBitsExp" : 4,
        "MaxAbsStride3D" : 0,
        "MaxAbsStride1D" : 1,
        "IndirectStride2DStream" : true,
        "AtomicOperations" : [ "Add", "Sub", "Min", "Max" ],
        "NumIdxUnitBitsExp" : 4,
        "MaxAbsDeltaStride2D" : 0,
        "LinearStride2DStream" : true,
        "MaxLength2D" : 2147483646,
        "MaxAbsStretch2D" : 1073741822,
        "nodeType" : "DirectMemoryAccess",
        "supportBuffet" : false,
        "numMemUnitBitsExp" : 4,
        "MaxAbsStretch3D1D" : 0,
        "IndirectIndexStream" : true,
        "NumStride2DUnitBitsExp" : 4,
        "writeWidth" : 32,
        "MaxAbsStride2D" : 1073741822,
        "numPendingRequest" : 16,
        "readWidth" : 32,
        "nodeId" : 0,
        "streamStated" : true,
        "numSpmBank" : 0,
        "IndirectLength1DStream" : true,
        "MaxAbsDeltaStretch2D" : 0
    }
    },
    "InputVectorPort.1" : {
    "dsagen2.sync.config.SyncKeys$IVPNode$" : {
        "vpImpl" : 0,
```


(continues on next page)

(continued from previous page)


```
        "vpStated" : true,
        "parameterClassName" : "dsagen2.sync.config.IVPNodeParameters",
        "depthByte" : 2,
        "nodeType" : "InputVectorPort",
        "repeatedIVP" : true,
        "nodeId" : 1,
        "broadcastIVP" : true
    }
    },
    "InputVectorPort.0" : {
    "dsagen2.sync.config.SyncKeys$IVPNode$" : {
        "vpImpl" : 0,
        "vpStated" : true,
        "parameterClassName" : "dsagen2.sync.config.IVPNodeParameters",
        "depthByte" : 2,
        "nodeType" : "InputVectorPort",
        "repeatedIVP" : true,
        "nodeId" : 0,
        "broadcastIVP" : true
    }
    },
    "OutputVectorPort.1" : {
    "dsagen2.sync.config.SyncKeys$OVPNode$" : {
        "vpImpl" : 0,
        "discardOVP" : true,
        "vpStated" : true,
        "parameterClassName" : "dsagen2.sync.config.OVPNodeParameters",
        "taskOVP" : true,
        "depthByte" : 2,
        "nodeType" : "OutputVectorPort",
        "nodeId" : 1
    }
    },
    "OutputVectorPort.0" : {
    "dsagen2.sync.config.SyncKeys$OVPNode$" : {
        "vpImpl" : 0,
        "discardOVP" : true,
        "vpStated" : true,
        "parameterClassName" : "dsagen2.sync.config.OVPNodeParameters",
        "taskOVP" : true,
        "depthByte" : 2,
        "nodeType" : "OutputVectorPort",
        "nodeId" : 0
    }
    }
},
"DSAGenEdges" : [ {
    "SourceNodeType" : "DirectMemoryAccess",
    "SourceNodeId" : 0,
    "SourceIndex" : 0,
    "SinkNodeType" : "InputVectorPort",
    "SinkNodeId" : 0,
```


(continues on next page)

(continued from previous page)


```
    "SinkIndex" : 0
}, {
    "SourceNodeType" : "InputVectorPort",
    "SourceNodeId" : 0,
    "SourceIndex" : 0,
    "SinkNodeType" : "Switch",
    "SinkNodeId" : 0,
    "SinkIndex" : 0
}, {
    "SourceNodeType" : "OutputVectorPort",
    "SourceNodeId" : 0,
    "SourceIndex" : 0,
    "SinkNodeType" : "DirectMemoryAccess",
    "SinkNodeId" : 0,
    "SinkIndex" : 0
}, {
    "SourceNodeType" : "Switch",
    "SourceNodeId" : 0,
    "SourceIndex" : 0,
    "SinkNodeType" : "Switch",
    "SinkNodeId" : 1,
    "SinkIndex" : 0
}, {
    "SourceNodeType" : "DirectMemoryAccess",
    "SourceNodeId" : 0,
    "SourceIndex" : 1,
    "SinkNodeType" : "InputVectorPort",
    "SinkNodeId" : 1,
    "SinkIndex" : 0
}, {
    "SourceNodeType" : "ScratchpadMemory",
    "SourceNodeId" : 0,
    "SourceIndex" : 0,
    "SinkNodeType" : "InputVectorPort",
    "SinkNodeId" : 0,
    "SinkIndex" : 1
}, {
    "SourceNodeType" : "OutputVectorPort",
    "SourceNodeId" : 0,
    "SourceIndex" : 1,
    "SinkNodeType" : "ScratchpadMemory",
    "SinkNodeId" : 0,
    "SinkIndex" : 0
}, {
    "SourceNodeType" : "OutputVectorPort",
    "SourceNodeId" : 1,
    "SourceIndex" : 0,
    "SinkNodeType" : "DirectMemoryAccess",
    "SinkNodeId" : 0,
    "SinkIndex" : 1
}, {
    "SourceNodeType" : "InputVectorPort",
```


(continues on next page)

(continued from previous page)

```
    "SourceNodeId" : 0,
    "SourceIndex" : 1,
    "SinkNodeType" : "Switch",
    "SinkNodeId" : 1,
    "SinkIndex" : 1
}, {
    "SourceNodeType" : "Switch",
    "SourceNodeId" : 0,
    "SourceIndex" : 1,
    "SinkNodeType" : "Switch",
    "SinkNodeId" : 2,
    "SinkIndex" : 0
}, {
    "SourceNodeType" : "ScratchpadMemory",
    "SourceNodeId" : 0,
    "SourceIndex" : 1,
    "SinkNodeType" : "InputVectorPort",
    "SinkNodeId" : 1,
    "SinkIndex" : 1
}, {
    "SourceNodeType" : "GenerateEngine",
    "SourceNodeId" : 0,
    "SourceIndex" : 0,
    "SinkNodeType" : "InputVectorPort",
    "SinkNodeId" : 0,
    "SinkIndex" : 2
}, {
    "SourceNodeType" : "OutputVectorPort",
    "SourceNodeId" : 0,
    "SourceIndex" : 2,
    "SinkNodeType" : "GenerateEngine",
    "SinkNodeId" : 0,
    "SinkIndex" : 0
}, {
    "SourceNodeType" : "OutputVectorPort",
    "SourceNodeId" : 1,
    "SourceIndex" : 1,
    "SinkNodeType" : "ScratchpadMemory",
    "SinkNodeId" : 0,
    "SinkIndex" : 1
}, {
    "SourceNodeType" : "Switch",
    "SourceNodeId" : 0,
    "SourceIndex" : 2,
    "SinkNodeType" : "ProcessingElement",
    "SinkNodeId" : 0,
    "SinkIndex" : 0
}, {
    "SourceNodeType" : "InputVectorPort",
    "SourceNodeId" : 1,
    "SourceIndex" : 0,
    "SinkNodeType" : "Switch",
```

(continues on next page)

(continued from previous page)


```
    "SinkNodeId" : 2,
    "SinkIndex" : 1
}, {
    "SourceNodeType" : "RecurrenceEngine",
    "SourceNodeId" : 0,
    "SourceIndex" : 0,
    "SinkNodeType" : "InputVectorPort",
    "SinkNodeId" : 0,
    "SinkIndex" : 3
}, {
    "SourceNodeType" : "GenerateEngine",
    "SourceNodeId" : 0,
    "SourceIndex" : 1,
    "SinkNodeType" : "InputVectorPort",
    "SinkNodeId" : 1,
    "SinkIndex" : 2
}, {
    "SourceNodeType" : "OutputVectorPort",
    "SourceNodeId" : 0,
    "SourceIndex" : 3,
    "SinkNodeType" : "RecurrenceEngine",
    "SinkNodeId" : 0,
    "SinkIndex" : 0
}, {
    "SourceNodeType" : "OutputVectorPort",
    "SourceNodeId" : 1,
    "SourceIndex" : 2,
    "SinkNodeType" : "GenerateEngine",
    "SinkNodeId" : 0,
    "SinkIndex" : 1
}, {
    "SourceNodeType" : "Switch",
    "SourceNodeId" : 1,
    "SourceIndex" : 0,
    "SinkNodeType" : "ProcessingElement",
    "SinkNodeId" : 0,
    "SinkIndex" : 1
}, {
    "SourceNodeType" : "RecurrenceEngine",
    "SourceNodeId" : 0,
    "SourceIndex" : 1,
    "SinkNodeType" : "InputVectorPort",
    "SinkNodeId" : 1,
    "SinkIndex" : 3
}, {
    "SourceNodeType" : "OutputVectorPort",
    "SourceNodeId" : 0,
    "SourceIndex" : 4,
    "SinkNodeType" : "RegisterEngine",
    "SinkNodeId" : 0,
    "SinkIndex" : 0
}, {
```


(continues on next page)

(continued from previous page)


```
    "SourceNodeType" : "OutputVectorPort",
    "SourceNodeId" : 1,
    "SourceIndex" : 3,
    "SinkNodeType" : "RecurrenceEngine",
    "SinkNodeId" : 0,
    "SinkIndex" : 1
}, {
    "SourceNodeType" : "Switch",
    "SourceNodeId" : 2,
    "SourceIndex" : 0,
    "SinkNodeType" : "ProcessingElement",
    "SinkNodeId" : 0,
    "SinkIndex" : 2
}, {
    "SourceNodeType" : "OutputVectorPort",
    "SourceNodeId" : 1,
    "SourceIndex" : 4,
    "SinkNodeType" : "RegisterEngine",
    "SinkNodeId" : 0,
    "SinkIndex" : 1
}, {
    "SourceNodeType" : "Switch",
    "SourceNodeId" : 3,
    "SourceIndex" : 0,
    "SinkNodeType" : "ProcessingElement",
    "SinkNodeId" : 0,
    "SinkIndex" : 3
}, {
    "SourceNodeType" : "Switch",
    "SourceNodeId" : 2,
    "SourceIndex" : 1,
    "SinkNodeType" : "Switch",
    "SinkNodeId" : 3,
    "SinkIndex" : 0
}, {
    "SourceNodeType" : "Switch",
    "SourceNodeId" : 3,
    "SourceIndex" : 1,
    "SinkNodeType" : "OutputVectorPort",
    "SinkNodeId" : 0,
    "SinkIndex" : 0
}, {
    "SourceNodeType" : "Switch",
    "SourceNodeId" : 1,
    "SourceIndex" : 1,
    "SinkNodeType" : "Switch",
    "SinkNodeId" : 3,
    "SinkIndex" : 1
}, {
    "SourceNodeType" : "Switch",
    "SourceNodeId" : 2,
    "SourceIndex" : 2,
```


(continues on next page)

(continued from previous page)

```
    "SinkNodeType" : "OutputVectorPort",
    "SinkNodeId" : 0,
    "SinkIndex" : 1
}, {
    "SourceNodeType" : "Switch",
    "SourceNodeId" : 3,
    "SourceIndex" : 2,
    "SinkNodeType" : "Switch",
    "SinkNodeId" : 2,
    "SinkIndex" : 2
}, {
    "SourceNodeType" : "InputVectorPort",
    "SourceNodeId" : 1,
    "SourceIndex" : 1,
    "SinkNodeType" : "Switch",
    "SinkNodeId" : 3,
    "SinkIndex" : 2
}, {
    "SourceNodeType" : "Switch",
    "SourceNodeId" : 1,
    "SourceIndex" : 2,
    "SinkNodeType" : "OutputVectorPort",
    "SinkNodeId" : 1,
    "SinkIndex" : 0
}, {
    "SourceNodeType" : "Switch",
    "SourceNodeId" : 3,
    "SourceIndex" : 3,
    "SinkNodeType" : "Switch",
    "SinkNodeId" : 1,
    "SinkIndex" : 2
}, {
    "SourceNodeType" : "ProcessingElement",
    "SourceNodeId" : 0,
    "SourceIndex" : 0,
    "SinkNodeType" : "Switch",
    "SinkNodeId" : 3,
    "SinkIndex" : 3
}, {
    "SourceNodeType" : "Switch",
    "SourceNodeId" : 0,
    "SourceIndex" : 3,
    "SinkNodeType" : "OutputVectorPort",
    "SinkNodeId" : 1,
    "SinkIndex" : 1
}, {
    "SourceNodeType" : "Switch",
    "SourceNodeId" : 1,
    "SourceIndex" : 3,
    "SinkNodeType" : "Switch",
    "SinkNodeId" : 0,
    "SinkIndex" : 1
```

(continues on next page)

(continued from previous page)

```
}, {
    "SourceNodeType" : "Switch",
    "SourceNodeId" : 2,
    "SourceIndex" : 3,
    "SinkNodeType" : "Switch",
    "SinkNodeId" : 0,
    "SinkIndex" : 2
} ]
}
```

Visualized, this adg would look like the following:

CHAPTER

SEVEN

SPATIAL SCHEDULER

This library contains tools to describe, model, and compile/schedule for spatial architectures. The scheduler acts as an in-between, mapping software programs to decoupled-spatial hardware accelerators.

7.1 Usage Overview

The scheduler is run through by using the *ss_sched* command. To run the scheduler by default run:

```
ss_sched [dfg-file] [adg-file] [options]
```

Optionally, the scheduler includes different flags that can help with compilation. We list these below:

- *-v* or *–verbose*

  Makes the logging more verbose, providing in-depth information about the scheduler's progress. This defaults to *False*.

- *-x* or *–design-explore*

  Enables Design-Space Exploration (DSE) for the scheduler. See Design Space Exploration.This defaults to *False*.

- *-f* or *–fpga*

  Assumes model is using the FPGA-based hardware. Defaults to using ASIC-based hardware.

- *-p* or *–print-bitstream*

  Dumps the binary bitstream upon successful scheduling. This defaults to *False*.

- *-t* or *–timeout*

  Kills scheduling if the process takes longer, in seconds, than the timeout. This defaults to *86400* or 24 hours.

- *-m* or *–max-iters*

  Maximum scheduling iterations. Oftentimes, the scheduler will reach this before the timeout. This defaults to *20000*.

- *-e* or *–seed*

  Sets the random seed for the scheduler. This defaults to a random value.

- *–dse-timeout*

  Sets the timeout for the DSE process, in seconds. This defaults to *-1* or no timeout.

- *-w* or *–sched-workers*

The number of workers used during scheduling. Workers schedule different dfg files in parallel. Helpful for when the Design-Space Exploration is enabled. Defaults to *1*.

- *-h* or *–help*

  Prints the help message.

7.1.1 DFG Model

By just supplying the DFG file, the scheduler can print the resulting dataflow graph visualization. For instance, to get a dfg visualization for *workload.dfg* run:

```
ss_sched workload.dfg
```

7.1.2 ADG Model

By just supplying the ADG file, the scheduler can both print the graph and estimate the single-core power/area/resources, depending on whether the FPGA flag is set.

For instance, to get the single-core estimated resource breakdown of the ADG file *adg.json*:

```
ss_sched adg.json -f
```

The schedule will also provide a dot file that can be used to visualize results. However, we recommed using the visualization script described here to get a better visualization.

7.2 Spatial Mapping Algorithm

The spatial scheduler workflow is described in Figure 1. In the first step, the spatial scheduler first generates a list of possible candidate placements for each dataflow node. These candidate mappings must follow certain requirements, like ensuring proper bitwidth alignment and the processing element having the correct functions. If any dataflow node does not have any possible candidates, we can safely terminate the scheduling process, as there exists no mapping that works for this dataflow graph.

The spatial scheduler then moves to routing, placing all possible combinations and attempting to find the best possible mapping spot. The spatial scheduler uses Dijkstra's algorithm to find the shortest path between nodes, creating distinct datapath's for each edge of the dataflow graph. To evaluate a candidate node mapping's effectiveness, the spatial scheduler evaluates an objective function, measuring the schedule's overall performance. If multiple candidate nodes have the same objective score, which frequently happens in practice, then the spatial scheduler will randomly choose a candidate to concretize. The routing process will continue until each dataflow node is mapped onto the spatial accelerator.

Following the routing process, the spatial scheduler calculates overprovisioning and latency factors to be used in the overall objective function. The latency factors are iteratively calculated by gradually tightening the latency bounds of each spatial node within a given edges route.

These factors result in an objective function, measuring a given schedule's performance, that is compared to prior iterations. If the objective function is described to be good enough (complete, no overprovisioning, sufficient memory access, and lack of latency violations), then the spatial scheduler returns this result. Otherwise, the spatial scheduler randomly unmaps different spatial nodes and repeats the previous steps, until the schedule either completes or fails due to time constraints.

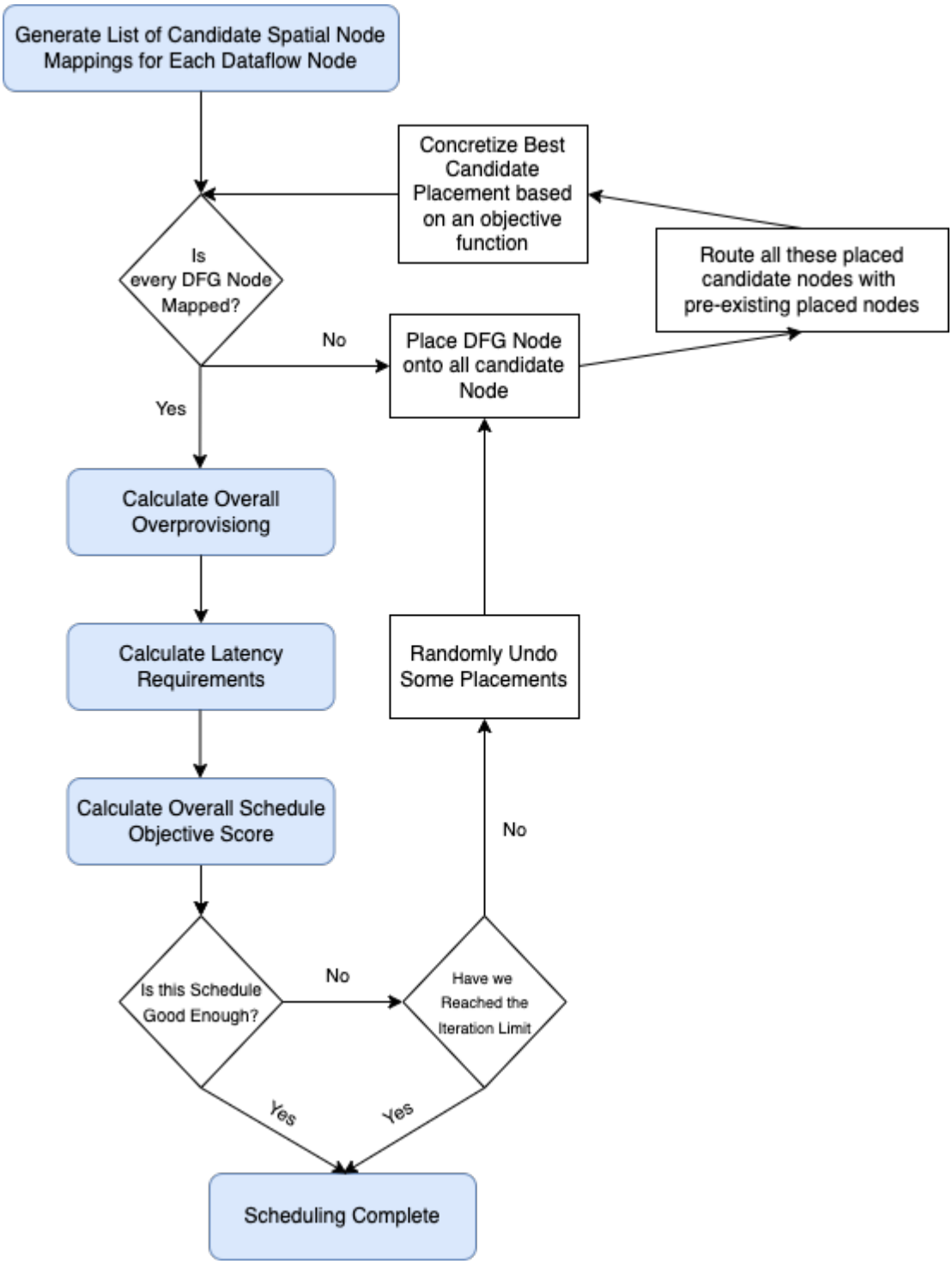


Fig. 1: **Figure 1:** The Spatial Scheduler Algorithm

7.2.1 Extra Capabilities

The spatial scheduler currently supports subnetwork and decomposible routing, allowing different software nodes of different sizes being able to schedule onto the same hardware node. This allows a greater exploration space, allowing the scheduler to find potential schedules without needing SIMD instructions. However, the bitstream generation doesn't currently support subnetwork or decomposible routing, although it will be supported in the future. Thus, utilize this feature at your own risk.

7.3 Spatial Mapping Rules

The following list contains the rules that are used to determine whether a given schedule is valid or not.

1. Vertex slots must be mapped to even slots.

   Example:

```
- Invalid:    | ___ | OPA | OPA | ___ |
- Valid:      | OPA | OPA | ___ | ___ |
- Valid:      | ___ | ___ | OPA | OPA |
```

2. Two DFG ports can not be mapped to a single VectorPort.
3. The stated dfg edge is always 8 bits wide.
4. A stated DFG port can not be mapped to a non-stated vectorport.
5. The stated dfg edge is always mapped to a stated VectorPort on bits [0-8]. The first link of a vector port only includes the stated edge.
6. The other links of the vectorport are statically assigned according to their value id. A InputPort value can not stradle two different links.
7. Memory DFG Edges, or those with either their source or destination vertex being a data node, can't utilize switches.

   Exampe:

```
- Invalid: SPM0 -> SWITCH0 -> IVP0
```

8. A dfg edge entering a non-switch must come in at an even slot

   Example:

```
- Invalid:    | ___ | OPA | OPA | ___ |
- Valid:      | OPA | OPA | ___ | ___ |
- Valid:      | ___ | ___ | OPA | OPA |
```

9. A switch can map to any contiguous slot.

   Example:

```
- Valid:      | ___ | OPA | OPA | ___ |
- Valid:      | OPA | OPA | ___ | ___ |
- Valid:      | OPA | ___ | ___ | OPA |
- Invalid:    | OPA | ___ | OPA | ___ |
```

10. A lower bitwidth edge must always be mapped to the lower bits of a granularity.

    Example:

```
A 16 bit edge mapped to a Node with granularity 32 and datawidth 64 bit granularity
- Valid: Mapping edge to bits: [0:16] or [32:48]
- Invalid: Mapping edge to bits: [16:32] or [48:64]
```

CHAPTER

EIGHT

DESIGN SPACE EXPLORER

8.1 Usage

The design space explorer is run through by using the *ss_sched* command, with the *-x* flag. To run the scheduler by default run:

Important: Currently the Design-Space Explorer only explorers fpga-based accelerators and doesn't support ASIC-based optimization. Thus it must be run with the *-f* flag.

The design-space explorer shares the same flags as the default scheduler.:

- *-e* or *–seed*

  Sets the random seed for the scheduler. This defaults to a random value.

- *–dse-timeout*

  Sets the timeout for the DSE process, in seconds. This defaults to *-1* or no timeout.

- *-w* or *–sched-workers*

  The number of workers used during scheduling. Workers schedule different dfg files in parallel. Defaults to *1*.

8.1.1 File Outputs

The design-space explorer outputs a number of files to the *vis* directory. The files are:

- *objectives.csv*

  A logfile of different dse-iterations. Useful for debugging and visualizing the design-space explorers progress.

- *final-schedadg.json*

  The produced adg file from the dse. Contains extra fields describing the system parameters and information about how the finalized schedules were mapped onto the accelerator.

- *prunned-schedadg.json*

  The prunned version of final-schedadg. Contains extra fields describing the system parameters and information about how the finalized schedules were mapped onto the accelerator.

- *iters* directory

A directory containing improved schedules. Each time the DSE improves the ADG, it will store that iterations ADG within this folder. This folder is useful for understanding what the DSE did at each iteration.

8.2 DSE Algorithm

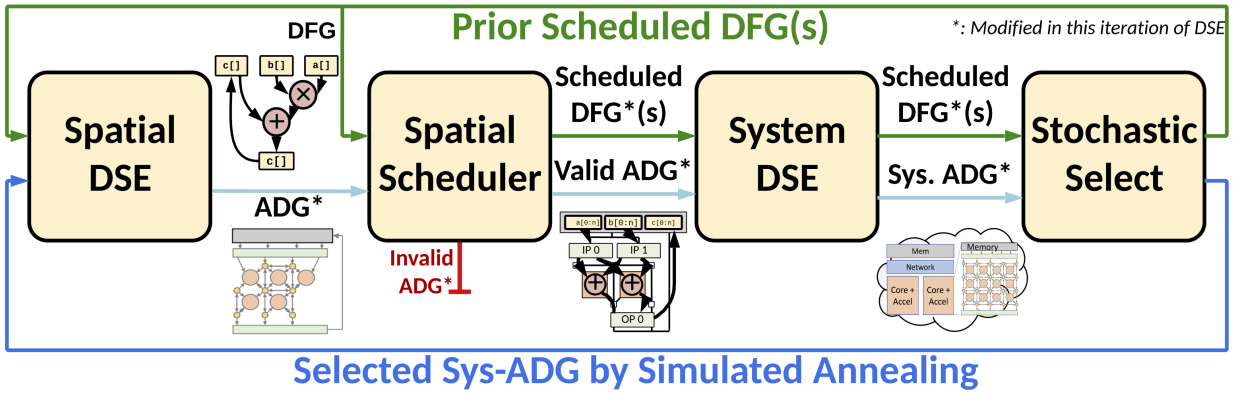


Fig. 1: **Figure 1:** DSE algorithm

The DSE algorithm is described within Figure 1. The algorithm works in several steps:

1. Spatial DSE

   In this step, the DSE modifies the inputed ADG. The algorithm randomly decides between adding, removing, or modifying different modular states. The number of modifications is determined by a temperature variable, which is set according to the iteration.

2. Spatial Scheduler

   After modifying the schedule, the scheduler then attempts to reschedule the modified DSE. If it fails to schedule or is overprovisioned, then the DSE iteration fails at this point and restarts using prior schedules.

3. System DSE

   During this step, the DSE uses the performance and resource models to determine the system parameters. The DSE fully explores the system parameters, chosing the design with the best performance and most cores, while remaining under full fpga utilization.

4. Stochastic Selection

   Finally, the dse stocastically chooses a new adg design based upon overall performance and single-core area. The DSE uses iteration number and temperature to determine probability of choosing a new design.

CHAPTER

NINE

RTL GENERATION

This is a guide to using the RTL generation framework.

9.1 Hardware Architecture Overview

This is a guide to using the RTL generation framework.

9.2 SoC Generation with DSA integrated via DSL/ADG

This is a guide to using the RTL generation framework.

9.3 Compile Verilator for RTL Simulation

This is a guide to using the RTL generation framework.

9.4 FPGA flow

This is a guide to using the RTL generation framework.

CHAPTER

TEN

WORKLOADS

Some placeholder text

CHAPTER

ELEVEN

API

11.1 DSA Scheduler API

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search