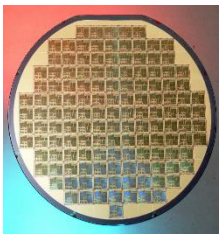
TECHNION – ISRAEL INSTITUTE OF TECHNOLOGY



VLSI SYSTEMS RESEARCH CENTER

VLSI LABORATORY

DEPARTMENT OF ELECTRICAL ENGINEERING DEPARTMENT OF COMPUTER SCIENCE

Design and Implement

CVU-Control Vector Unit for a VGIW Architecture

Final report

Submitted by:

Rabea Mahajna Kamal Khateb

Supervisor:

Dani Voitsechov

Contents

Abstract

In this work we present the implementation of the CVU- Control Vector Unit for the VGIW- vector graph instruction word- architecture. In previous Project we implemented Basic Block Scheduler, another architectural component. This book details the roles occupied by each component and how they interact with each other. The VGIW architecture is positioned as an energy efficient design alternative for GPGPUs. This architecture can be seen as an extension to another architecture known as SGMF- Single-Graph Multiple-Flows. The SGMF architecture maps a complete compute kernel (program code) represented as control-dataflow graph, onto a coarse-grain reconfigurable fabric. While the VGIW architecture maps basic blocks represented as dataflow graphs.

For the VGIW architecture, the mapping process is based on the representation of the kernel as several sub-graphs. The kernel is divided to smaller code blocks called Basic Blocks –BB. These blocks have single entry and single exit. Each BB is mapped separately to the fabric.

The BB scheduler is responsible for managing the scheduling of these BBs. The BB scheduler determines in which order the mapping of the BBs should be done. It's also responsible for keeping and updating data the status of each BB during run kernel time.

While the BB scheduler preserves the machine state at kernel run time, the CVU is responsible for supplying the BB scheduler with the data needed for that purpose.

1. Introduction

This book presents and details our work on designing and implementing CVU- Control Vector Unit for VGIW architecture. We present the background from which the necessity to our design has emerged. Generally the project deals with the implementation of CVU- Control Vector Unit which is an integral part of a GPGPU alternative architecture known as VGIW architecture that is suggested by our supervisor Dani Voitsechov. The first part of the book we explain the background of our design, view some basic terms that are essential for having a clear view of the project. Then we show two alternative architectures, SGMF and VGIW. We explain how these two architectures operate, while focusing on how they relate to our project. We focus on the main differences between SGMF and VGIW. And explain how these differences create the need to our design which is needed in VGIW architecture. The second part of the book is dedicated to the design itself, first we make an abstract view of the design from previous project, explain he functionality that it provides, then we demonstrate how this project extends this functionality, and how the two components interact with each other. The last section of this part is dedicated for a full review of the CVU, we go over the units that build it up, explain the functionality that each unit support and how the different units interact with each other to supply the desired functionality of the design. We view an example for a better understanding of the process. Then we view the simulations we ran on the design and some of the implementation details that are shown in the wave view.

2. Project Background

2.1 A brief on GPGPUs

General purpose computing on graphics processing units is a methodology[1] for high-performance computing that uses graphics processing units to crunch data commonly called GPGPU (General Purpose Graphic Unit). The characteristics of these graphical units have enabled the development of high-performance general purpose graphics processors. GPs achieve high performance and high parallelism level by incorporating hundreds of relatively simple processing units to operate on many data elements simultaneously. And when dealing with big amount of data, there will be plenty to operate on in parallel and by this high throughput is achieved.

GPU properties lead to a very different processor architecture from traditional CPUs. CPUs devote a lot of resources (primarily chip area) to make single streams of instructions run fast, including caching to hide memory latency and complex instruction stream processing (pipelining, out-of-order execution and speculative execution). GPUs, on the other hand, use the chip area for hundreds of individual processing elements that execute a single instruction stream on many data elements simultaneously. Memory latency is hidden by very fast context switching; when a memory fetch is issued while processing one subset of data elements, that subset is set aside in favor of another subset that is not waiting on a memory reference.

GPGPU technology comes very handy in fields that rely on large data processing such as: computer vision and video and image processing, physical simulation applications, linear algebra operations etc.

2.2 Basic terminology

In this section we go through few basic terms and explain them in order to use them in the upcoming sections of this report.

2.2.1 Control-flow Graph

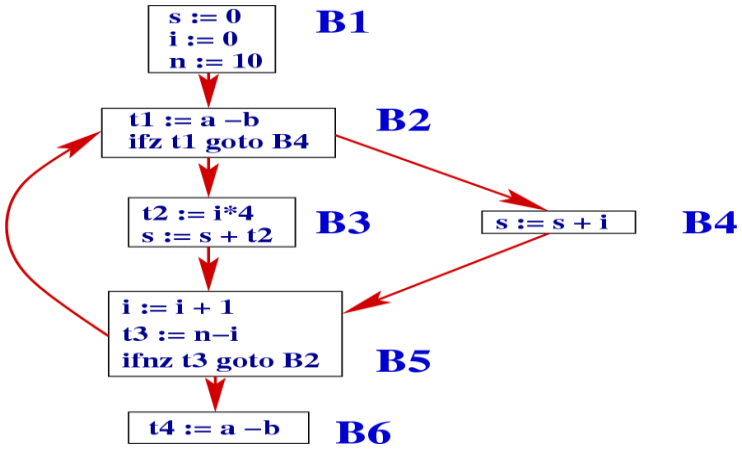


Figure 1: Control-flow graph. the graph presents the programgram as a sequnce of commands with all the execution paths possible for the program.

[1] www.tacc.utexas.edu/documents/13601/88790/8Things.pdf

In computer science, control flow[2] (or alternatively, flow of control) is the order in which individual statements, instructions or function calls of an program are executed or evaluated. The execution of an instruction is determined by the outcome of previous control statement that determine which control path should the program take. The example in Figure 1 demonstrates this abstraction, the outcome of the branch command at the end of B2 (ifz t1 goto B4) for example determines the next instruction to be executed. Any valid walk through the graph can be an actual run path of the program described in the figure.

This abstract representation highlight the order of executing the instruction and makes it easy to keep a track on the program progress. Some of The most common architectures for processors Implements this abstraction such as the von-Neumann architecture.

2.2.2 Dataflow Graph

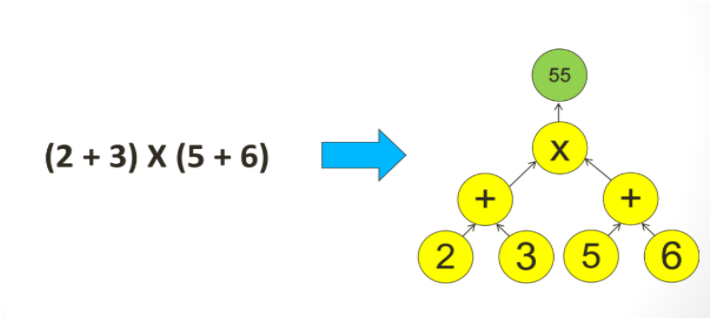


Figure 2: Dataflow graph. The starting point are the operands (Data elements). The graph highlightes the data depencecies between these operands.

Dataflow graphs are another way of abstract representation of a programs. Computer Architectures that rely on this abstraction have a different design from the traditional Von-Neumann architecture. The execution of instructions is solely determined based on the availability of input arguments to the instructions, so that the order of instruction execution is unpredictable. Figure 2 shows how the operands are fed at the starting point of the flow (the bottom of the graph in figure 2) and intermediate compute results are passed through the functional units until the final result is supplied at the end of the flow.

Data-flow graphs highlight the data dependencies in a given program. They also supply a better view of the program functionality since commands are related to each to other based on the data dependencies.

2.2.3 Basic Block

In this project often use the term Basic Block (BB). Basic block is a sequence of instructions that have a start label (first instruction in the sequence) and ends with a conditional or unconditional branch. Basic block has a single entry single exit structure, meaning the sequence is executed serially without any branching in between. Figure 1 we can see several BBs within the same program that are connected via control statements. In Figure1 the BBs are labeled as B1, B2, B3 …

[2] https://en.wikipedia.org/wiki/Control_flow

2.3 Von-Neumann architecture

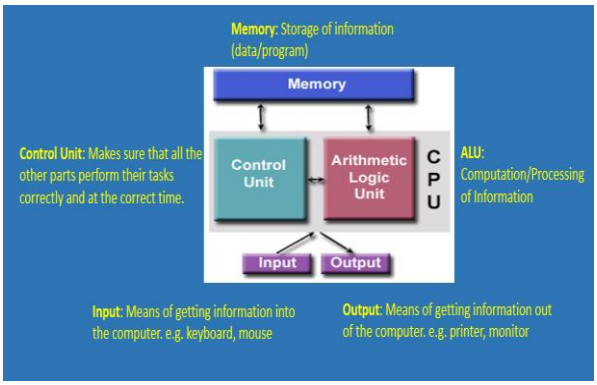


Figure 3: overview of von-Neumann machine.

Von Neumann computer architecture is the most common architecture used in processors industry. The is built of three major parts central processing unit (CPU), memory, and input/output devices (I/O).These three components are connected together using the *system bus*. Von-Neumann machines execute programs as sequence of instructions. The sequence is determined during run time. The machine design relies on the control flow graph representation of programs. Thus the sequence of instructions is actually a specific control path along the graph that the program takes.

The control unit is responsible for fetching instructions from the memory that is shared for data and instruction. It supplies the ALU with the control signals to perform the instruction correctly. ALU write/read the computation results to the memory. The program can have input from outside input device it also can redirect its output to external output devices. Figure 3 provides an abstract view of how components interact with each other. The controller as said fetch the instructions in the correct order this made by a special register called PC (program counter or IP-Instruction pointer). This register holds the address of the next instruction to fetch. Most of the time the instructions are fetched serially from the memory so the PC is incremented by the size of one instruction. When there is a branch the PC is loaded with the right destination according to the branch outcome. The control path of the program is determined during, and this sets the order of executing instruction. According to this von-Neumann machines makes it easier to track the program progress. This model is also preferable due to its simplicity. On the other hand, data and computation result are loaded/stored from and to the memory and the register file. This means that data dependencies are resolved through repeated accesses to these two memory devices which harms the machine performance, power efficiency and the ability to parallelize the program.

The von-Neumann architecture[3] has been incredibly successful, with most modern computers following the idea. You will find the CPU chip of a personal computer holding a control unit and the arithmetic logic unit (along with some local memory) and the main memory is in the form of RAM sticks located on the motherboard.
But there are some basic problems with it. Because of these problems, other architectures have been developed. These alternatives will be discussed later.

[3] http://www.c-jump.com/CIS77/CPU/VonNeumann/lecture.html#V77_0010_von_neumann

3. Alternative architectures for GPGPUs

3.1 SGMF Architecture

The single-graph multiple-flows (SGMF) is an alternative architecture for GPGPUs that combines coarse-grain reconfigurable computing with dynamic dataflow to deliver massive thread-level parallelism. The CUDA-compatible SGMF architecture is positioned as an energy efficient design. The architecture maps a compute kernel, represented as a dataflow graph, onto a coarse-grain reconfigurable fabric composed of a grid of interconnected functional units. Each unit dynamically schedules instances of the same static instruction originating from different CUDA threads. The dynamically scheduled functional units enable streaming the data of multiple threads (or graph flows, in SGMF parlance) through the grid. The combination of statically mapped instructions and direct communication between functional units obviate the need for a full instruction pipeline and a centralized register file, whose energy overheads burden GPGPUs.

The proposed single-graph multiple-flows[4] (SGMF) processor core is composed of a grid of interconnected functional units onto which compute kernels are mapped. An SGMF processor can be composed of multiple SGMF cores. The SGMF core employs the tagged-token dynamic dataflow model to concurrently execute multiple threads. Every thread is tagged, and the tag is attached to all in-flight data values associated with the thread. Tagging the values allows threads to bypass memory-stalled threads and execute out-of-order, while data-dependencies inside each thread are maintained. This increases overall utilization by allowing unblocked threads to use idle functional units. SGMF cores thus benefits from two types of parallelism: a) pipelining the instructions of individual threads through the CGRF; and b) simultaneous multi-threading (SMT) achieved by dynamically scheduling instructions and allowing blocked threads to be bypassed. Furthermore, the execution model enables the processor to fetch and decode instructions once, when the program is loaded, and directly communicates values between its functional units. The dynamically scheduled spatial design is more energy efficient than von-Neumann-based GPGPUs.

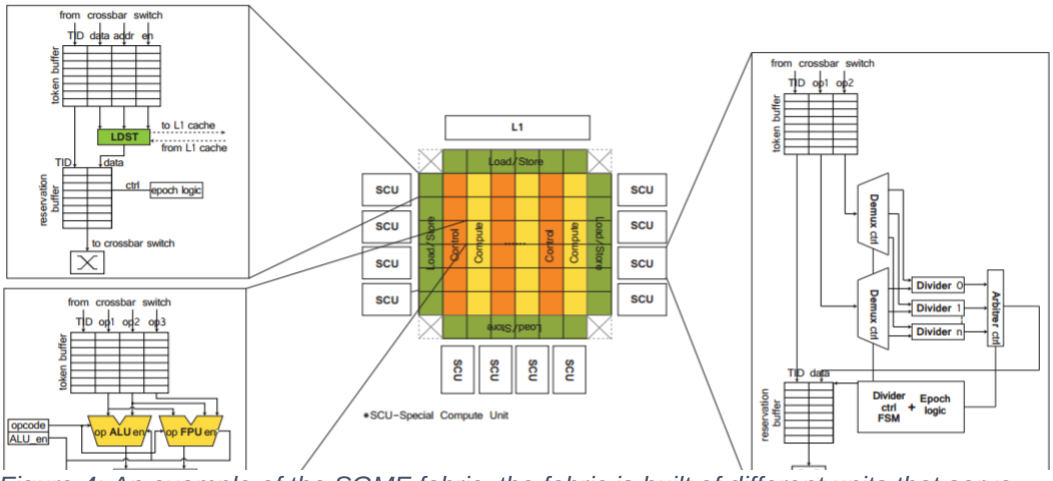


Figure 4: An example of the SGMF fabric. the fabric is built of different units that serve different purposes such as LD/STR, compute, control units.

[4] Single-Graph Multiple Flows: Energy Efficient Design Alternative for GPGPUs

The fabric in Figure 4 enables Different connectivity configurations between the units to match the functionality of the kernel.

Example for mapping between kernel and the fabric.

Let's assume we want to execute a simple kernel that does the following:

Func1 (a, b, c, x) {

Return (c + x + b)*(a + x);

}

The fabric should allocate enough compute units to perform the function. The functional unit's should be connected to match the kernel functionality.

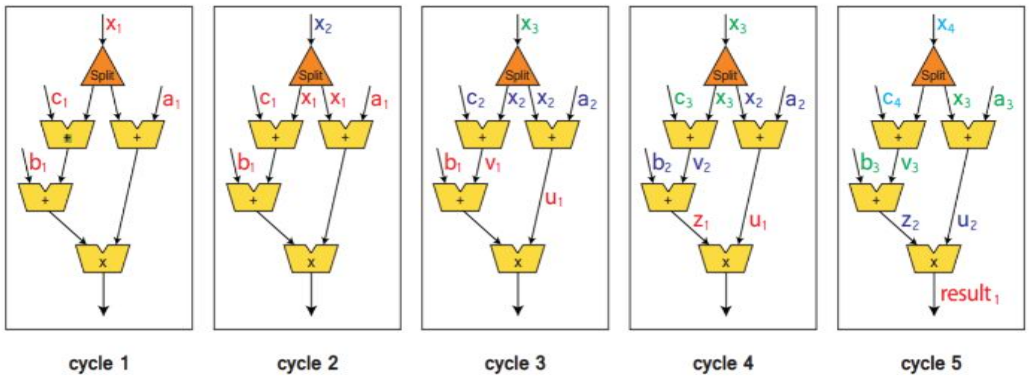


Figure 5: An example of specific configuration of the fabric units. the figure shows also how different data operands that origante from different threads are streamed through the same static units.

In Figure 5 we can see different units in the fabric connected in specific way to match the computational function. Once the mapping is done stream of data that originate from different threads are streamed through the fabric. Data operands are labeled with unique tag matching their thread.

The SGMF architecture, however, is limited by the static mapping of a kernel's CDFG to the MT-CGRF core and cannot execute large CUDA kernels efficiently. First, the limited capacity of the MT-CGRF core cannot host kernels whose CDFG is larger than the reconfigurable fabric. Second, as all control paths through a kernel's CDFG are statically mapped to the MT-CGRF, the core's functional units are underutilized when executing diverging control flows.

3.2 VGIW architecture

The hybrid dataflow/von Neumann vector graph instruction word (VGIW) is an alternative architecture for GPGPUs. This architecture is suggested as an improvement to SGMF. This hybrid architecture combines between the Von-Neumann control-flow architecture and the dataflow SGMF architecture. VGIW represents basic blocks as dataflow graphs (i.e., graph instruction words) and concurrently executes each block for a vector of threads using the MT-CGRF core. Meanwhile, threads' control flows determine the scheduling of basic blocks. This hybrid model [5]enables the architecture to dynamically coalesce all threads that are about to execute a basic block into a thread vector that is executed concurrently. The VGIW architecture thus preserves the performance and power efficiency provided by the dataflow MT-CGRF core, enjoys the generality of the von Neumann model for partitioning and executing large kernels, and eliminates inefficiencies caused by control flow divergence.

The idea of mapping kernels into a reconfigurable fabric as shown in SGMF is used in the VGIW architecture but instead of mapping the whole kernel at once the kernel is divided to smaller code blocks called Basic Blocks. The kernel is represented as a control flow graph connecting these Basic Block, while each basic block can be represented as a dataflow graph and can be mapped separately to the fabric.

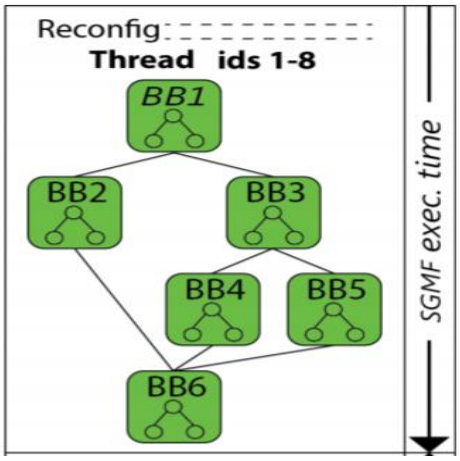


Figure 6: program mapping in SGMF. The complete kernel is mapped to the fabric.

[5] Control Flow Coalescing on a Hybrid Dataflow/von Neumann GPGPU

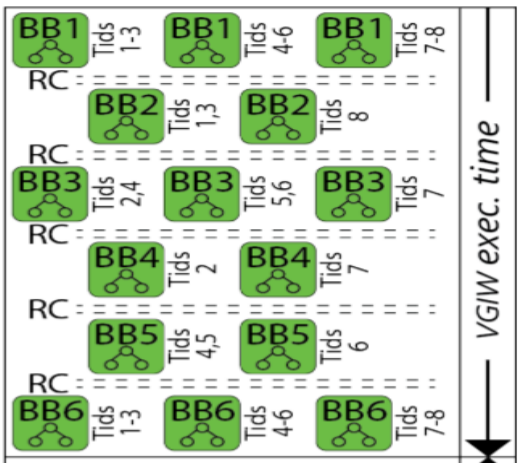


Figure 7: program mapping in VGIW. BBs are scheduled to run separately. Each BB is mapped separately. The fabric may create several replicas of the same BB for higher parallelism.

Figure 6 and Figure 7 highlight the main difference between SGMF and VGIW. The same kernel (program) is executed by both architectures. In figure 7 we can see that BBs are executed individually in different time slots while in figure 6 the threads running the kernel can run different BBs the same time. Breaking the kernel into BBs simplifies the mapping to the fabric. It also leads to higher utilization of the fabric units. In SGMF the fabric is divided between the different BBS and functional units are in idle state when there are no threads to run there matching BB while in VGIW the fabric contains several replicas of the same BB block and BB are scheduled only when there are threads waiting on them to run.

4. Implementation and design

4.1 Top-level model overview of the previous project.

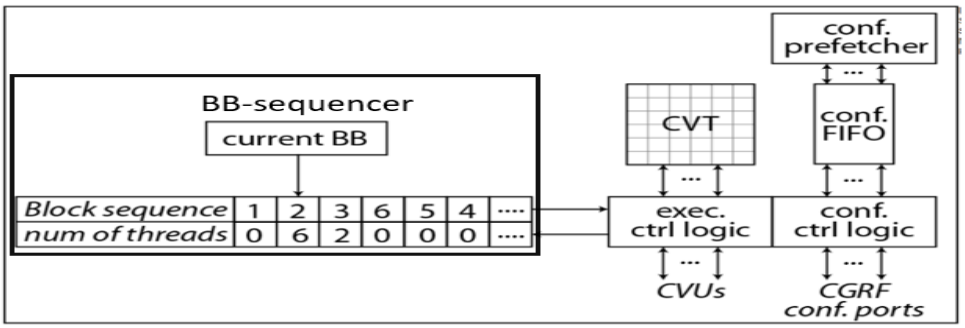


Figure 8: Top-model of the BB scheduler.

The BB Scheduler consists of three major units, CVT- control vector unit, BB-Sequencer, Exec-ctrl-logic as shown in figure 8. Chapters 4.1.1, 4.1.2, 4.1.3 and 4.2 explain how these units interact with each other and what duty each unit fulfills.

4.1.1 CVT-control vector unit

The CVT maintains a list of threads whose control flow has reached a basic block in the kernel. When that basic block is scheduled for execution and the MT-CGRF core has been con- figured with its dataflow graph, all its pending threads are streamed through the core. The table is updated dynamically. When the core finishes executing a thread's current basic block and its next basic block is determined, the core adds the thread's identifier to the destination block's entry in the table.

4.1.2 BB-sequencer

This unit implements the scheduling algorithm. The first BB to run is the entry point of the program. When all the threads running a specific basic block have retired this unit determine the next BB to run based on the status of the CVT.

4.1.2.1 Scheduling algorithm

We chose the scheduler algorithm according to the paper, as it shown we always choose the minimum order BB which has threads waiting to run it.

This pseudo code explains the scheduling algorithm.

```
BB getFirstBBReadyToRun(CVT){

For ( i=0; i< CVT.length ;i++ ){     -Scan all the CVT ,serial loop

        if(CVT[i] is ready to run )  -return the first BB which is ready to run

                return CVT[i];

        }

        return finished;   - Program is finished

  }
```

4.1.3 Execution-control-logic (exec. Ctrl-logic)

This unit is the controller of our component. It supplies all the control signals needed for the other components in order to function correctly. For example it determines when to enable the functionality of the BB-sequencer it's also responsible for managing reads and updates made to the CVT to keep it consistent with the kernel status.

4.2 BB scheduler Overall functionality

The basic block scheduler controls the kernel's execution by selecting the next basic block to execute, configuring the MT-CGRF core to run it, and sending thread IDs to the core. The scheduler also serves as a frontend to the CVT. The BBS reads basic block vectors from the CVT to send them to the core and, conversely, updates the block vectors in the CVT based on the resolved branch information it receives from the MT-CGRF [6]core. The finite size of the CVT imposes a limit on the number of threads that can be tracked at any given time. To execute a kernel, the runtime software loads its basic block sequence to the BBS, along with the per-block configuration of the MT-CGRF core. The runtime then signals the BBS to set all bits in the entry block vector (block ID 0). Finally, the BBS begins sending batches of thread IDs to the core for execution. Thread batches are sent as <base threadID, bitmap> tuples, where base thread ID represents the first thread in the batch (first bit in the bitmap), and the bitmap indicates which of the consecutively subsequent IDs are scheduled to execute the current basic block. Each packet consists of a 16-bit thread ID and a 64-bit bitmap. When the BBS sends a thread batch packet to the core, it zeros the corresponding bits in the CVT. Conversely, when a batch of threads finish a basic block, the core sends the BBS two batch packets, one for each of the block's successors. The BBS updates the CVT by OR-ing the bitmaps received from the core with the existing data in the CVT. An OR operation is required since a block may be reached by multiple control flows. During the execution of a block, the BBS pre-fetches the configurations of the following blocks into a con- figuration FIFO. Once the execution of all threads in the current block completes, a reset signal is sent to the nodes in the grid. The reset clears the nodes and configures the switches to pass tokens from left to right. The BBS then feeds the configuration tokens to the grid from its left perimeter.

4.2.1 Runtime example

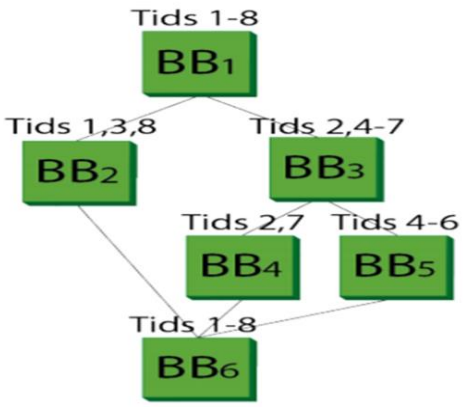


Figure 9: compute Kernel represented as control-data-flow graph. Threads 1-8 run the kernel. Each thread is assigned a unique Tid. Each thread takes its own execution path.

[6] Control Flow Coalescing on a Hybrid Dataflow/von Neumann GPGPU

The kernel is represented in graph 9. Each node is a BB and edges between nodes are branch commands. Based on the outcome of the branch the control path is determined per thread. The kernel has a starting point (BB1) that all the threads start by executing it, and an exit point (BB6 in this example) that the threads running the kernel execute on finish.

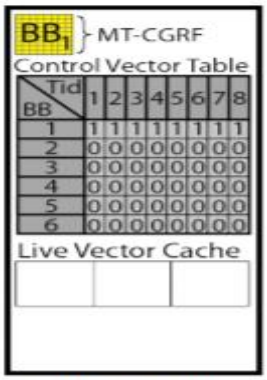


Figure 10: Initial state of the CVT. All threads are waiting on BB1.

Stage 1: all the threads are waiting on BB1 as shown in Figure 1 (the first row is set to 1'). The MT-CGRF is configured according to BB1. The table shown in this figure represents the CVT. For each BB there is a matching row that specifies which threads are waiting on the BB to run. Bits that are set to '1' indicates that threads with IDs matching the column number are waiting on the BB. Batches of threads are launched gradually according to the MT-CGRF capacity.

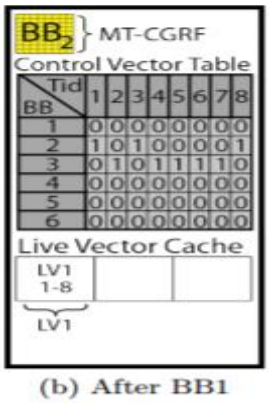
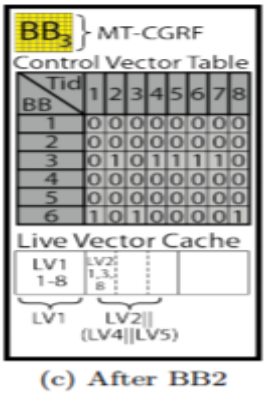


Figure 11: CVT state after BB1 execution is done.

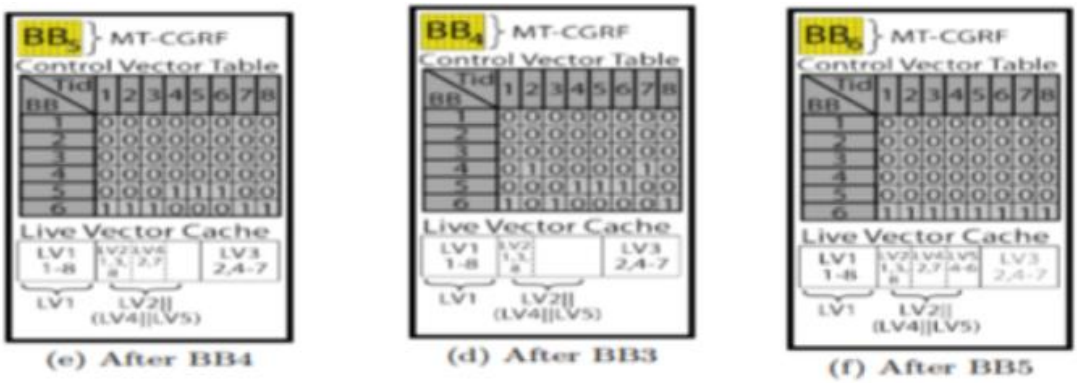
Stage 2: all the threads have executed BB1. The bit matching each thread is set to '1' in the destination BB for each thread. And thus we keep the CVT updated during runtime. Thread 1, 3, 8 have BB2. Threads 4, 5, 6, 7 reached BB3. At this point BB-sequencer is enabled to choose the next BB to run. BB2 is the next to run.



(c) After BB2

Figure 12: CVT state after BB2 execution is done.

Stage3: from this point on. All the stage are the same. After the chosen BB have finished to run. The table is updated with the destination of each thread. Then the next BB is chosen.



(e) After BB4 (d) After BB3 (f) After BB5

Figure 13: complete the run example presented above. after the execution of each BB. the CVT is updated.

4.3 CVU interaction with BB scheduler

Figure 13: BBs interface with CVU

In previous section we saw that access (read/write) to the CVT is done by 64 bits bandwidth. This approach shorts down the number of accesses made to the CVT, since we release/write a bunch of threads at each access. Eventually these threads are streamed separately through the fabric, so thread released from the same batch may take different paths and need to be written back to different BB targets, which generates a new CVT state.

How is this related to previous project?

As described before, the received batch from the CVT contain the threads that are waiting to run the BB which was chosen by the BB-Scheduler, from now on we will refer to this process as threads initiating.

On the other hand, the CVU receives thread that finished their run through the BB flow, and they are ready to be written to different targets according to their control flow.

The CVU re-join these threads into 64 bit width batches so they can be written back to the CVT. This is how we keep the CVT updated with threads flow during run time. From now on we will refer to this process as threads terminating.

4.4 CVU design review

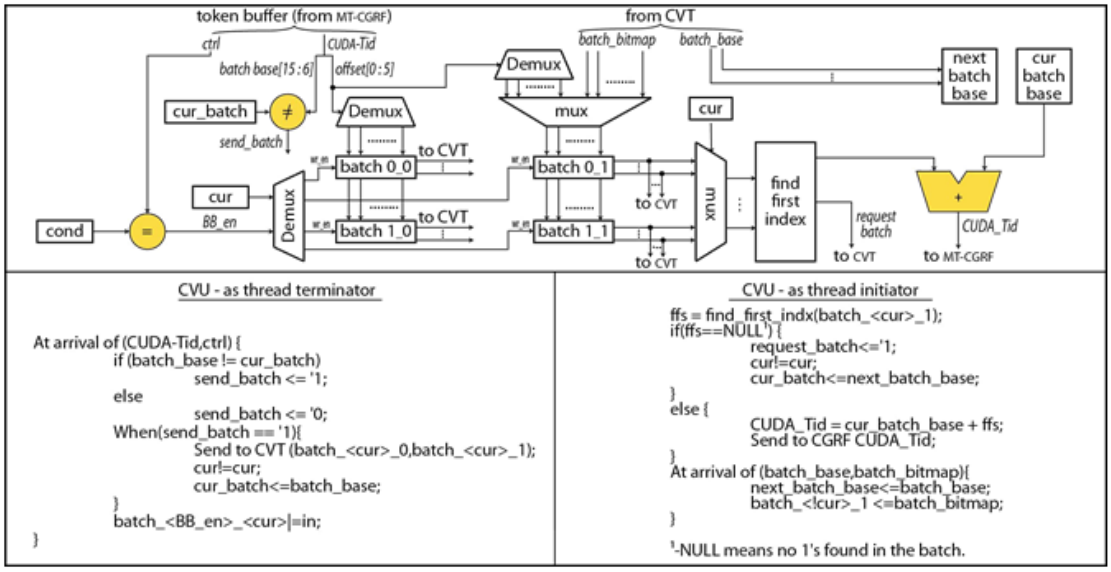


Figure 14: The control vector unit (CVU) when functioning as a thread initiator/terminator.

As said in previous section, each CVU can function both as a thread initiator and as a thread terminator (Figure 14). Each replica of a basic block's data graph is assigned an initiator CVU and a terminator CVU.

When functioning as a thread initiator, the CVU receives thread batches from the BBS. It then computes the CUDA ThreadIDX coordinates for the initiated threads and sends them to execute. When a basic block is replicated in the MT-CRGF (which is often used by the compiler to maximize the core's utilization), each replica is assigned an initiator CVU.

As described in section 4.3, thread batches are communicated as <basethreadID, bitmap> tuples. When a batch arrives, the CVU begins looping over the bitmap to identify the thread IDs it should initiate (by adding the set bits' indices to the base thread ID). To avoid stalls, CVUs use double buffering of thread batches. Whenever a batch is received, the CVU immediately requests the next batch from the BBS.

Conversely, when a CVU functions as a thread terminator, it executes the basic block's terminating branch instruction to determine the next basic block that the thread should execute. The destination block IDs (up to two) are stored in the CVU's configuration register. At runtime, the input token to the CVU determines which of the two targets should be executed next. The CVU maintains two thread batches, one for each possible tar-get block ID and adds each thread to the batch that corresponds with its branch outcome. Notably, since threads are executed out-of-order, thread IDs from different batches may be interleaved. To support threads that complete out-of-order, the CVU maintains a pair of thread batches for each destination block ID. Once the CVU encounters a thread ID from a different batch, it sends the current (possibly partial) batch to the BBS. In total, a CVU maintains storage for 2 batches. Each batch

is maintained using a 68-bit register (4-bit thread BaseID + 64-bit bitmap), for a total of 136 bits.

5. Simulations

This section presents wave views from different parts of the design. We describe more in detail how each components functions, what control signals participate in each process and how the design supplies the desired behavior.

CVU INITIATOR

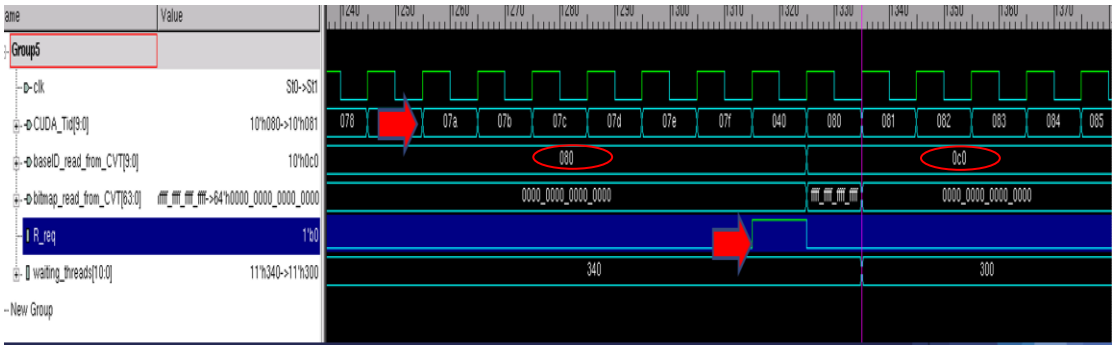


figure 15 , waves for CVU we functioning as initiator

As we can see, (Upper Arow) CUDA_TID pointing to the individual threads launching from received CVT's batches stored in CVU buffer .

These batch has it's own base_id which is now pointing to 0X080 .
the other arrow pointing to pose edge of read_request signal , which means we finished the given batch and initiated all the threads to the fabric , now we are launching a read request from the CVT to continue initiating the waiting threads of a given BB respectively.

We can see that the offset of the new batch changes (0X0C0) which means we are initiating new threads .

CVU TERMINATOR

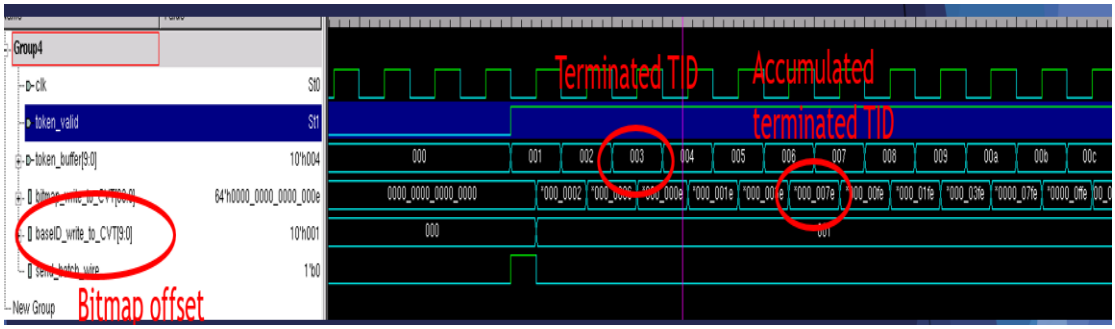


figure 16 , waves for CVU we functioning as terminator

As can be seen , we are collecting terminated threads IDs from the fabric and rejoining them into on batch consider their base ID.

CVU TERMINATOR WHEN SWITCHING BUFFERS

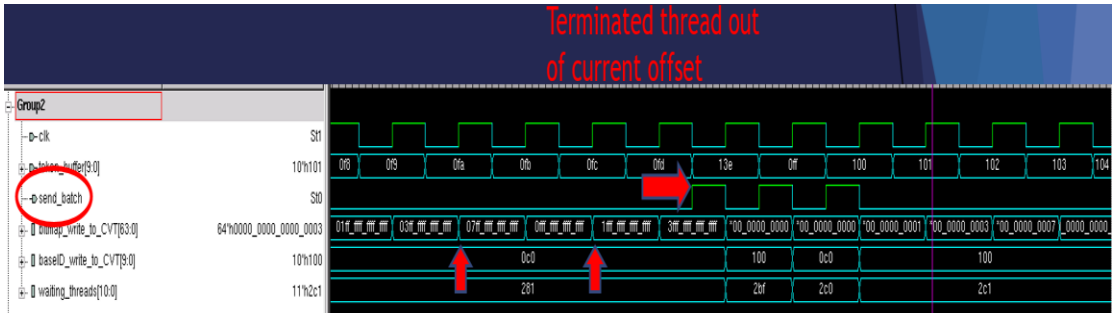


figure 17 , waves for CVU we functioning as terminator switching batches (buffers)

The vertical arrow points out the update of the batch with the terminated threads. The horizontal arrow is pointing the pose-edge of send batch, this happening due to the arrival of a new terminated threads that is out of the current base ID range.

CVU interacts with BB-SCHED

figure 18 , waves for CVU interacts with BB-SCHED , this shows the new project integration with old project

This wave form captures the interaction between the new unit CVU and the old unit BB-SCHED, this is done by integration of both project into one big project.

The first arrow points to the terminated threads id and shows the CVU functions as terminator, the second arrow points to the total number of collected (finished running) threads from the fabric and returned to the CVT, we can see when the number is 0X400 (which means 1024 in decimal our total MAXIMUM number of threads) so the CVT is now in a new state and the BB-SCHED unit is enabled to choose a new BB respectively to the new CVT state

And this can be seen in the third arrow when switching from BB 0 to BB 2,
the fourth arrow shows the new CVU function mode which is initiator , it launches a read request to the updated CVT ,the CVU is now initiating threads from BB=2 to the fabric and so on .

5.1 simulation reproduction

This chapter deals with practical details, and explains step by step how to reproduce the results shown above.

1) Step 1:
   The traces are supplied by the project supervisor based on real kernels run time traces. First we build an input file from these traces that's compatible our design. For this purpose we have used 2 Perl scripts. The first one runs over the trace and builds the control flow graph of the kernel, and allows us to know for each BB what its targets are. The second script produces the input file each trace event is translated to input vector and written down in a new line of the produced input file.
   The scripts name are:
   First script: log_parser_aux,
   Second script: log_parser_new
   The result shown above are based the trace file named:
   vgiw_inv_mapping.txt.

To produce the input we run the second script the trace file path:

```
C:\Users\kkhatib\Desktop\project_final>
C:\Users\kkhatib\Desktop\project_final>./log_parser_new ./vgiw_inv_mapping.txt
```

We run the first script implicitly by the second script. This command produces an input file name: vgiw_inv_mapping.txt_parsed

2) Step 2:
Once we produced the desired input file. We configure our test bench to use it as its input we simply change the path of the input file pointed to in the test bench to the desired input file. The test bench name is:
test_bench_new.sv

```
//*****************
module test_bench;

logic clk, rst;
logic [23:0] input_token, input_token_wire, last_poped;

//************OUTPUT*********
logic [9:0] last_initiated;
logic [31:0] output_token;
logic [63:0] bitmap_read_from_CVT;
logic [23:0] queue[$]={0};
integer data_file,scan_file; // file handler

`define NULL 0

initial begin
rst=1;
clk=0;
input_token = 22'b0;
data_file = $fopen("./vgiw_inv_mapping.txt_parsed", "r");
  if (data_file == `NULL) begin
    $display("data_file handle was NULL");
    $finish;
  end
```

The highlighted path in yellow is the path of the input file

3) Step 3:
Now we are ready to recompile the model the new input file configure. Then we run simulation wave view tool. To do so we simply run 2 commands shown below, the commands should be run from the directory that contains all the source files.

Command 1:
vcs -debug_all -debug –sverilog test_bench_new.sv TOP_model.sv RF_single.sv register_buffer.sv project.sv mux_2_1.sv mux_16_1.sv First_Index.sv Demux.sv dec_w_16_1.sv dec_16_1.sv CVU_init_kamal.sv CVU_CVT.sv CVT_17_10.sv ctrl_terminate.sv ctrl_read_mode.sv ctrl.sv count_ones.sv BB_seq_new.sv

Command 2:

simv -gui

After running command 2 the simulation wave view tool will be launched. And the results can be checked using the tool.

6. Synthesis

The synthesis is done using tower 0.18u library, the results in the VGIW paper are based on the 65 nm library. So we scaled our result to 65 nm technology to be able to compare the results:

Power Scaling (taken from: http://www.ece.utah.edu/~kstevens/6770/lecture-notes/scaling.pdf)

In ideal process scaling applied to power: (P = C*V^2*f)

- Capacitance = s
- Frequency = 1/s
- Voltage = s

So

$$P = S * S^2 * \frac{1}{S} = S^2$$

$$S = \frac{180}{65} \approx 3\,, \quad S^2 \approx 9$$

After scaling the results see that we are consistent with the results in the paper.

- Power

```
Global Operating Voltage = 1.8
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1pW


  Cell Internal Power  = 107.0896 mW   (52%)
  Net Switching Power  = 100.3100 mW   (48%)
                         ---------
Total Dynamic Power    = 207.3996 mW  (100%)

Cell Leakage Power     =   3.5247 uW

Information: report_power power group summary does not include estimated clock tree power. (PWR-789)

                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000              0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000              0.0000             0.0000  (   0.00%)
clock_network      0.0000           0.0000              0.0000             0.0000  (   0.00%)
register           0.0000           0.0000              0.0000             0.0000  (   0.00%)
sequential        89.9349           0.7146          2.3743e+06            90.6532  (  43.71%)
combinational     17.1551          99.6113          1.1508e+06           116.7449  (  56.29%)
--------------------------------------------------------------------------------------------------
Total            107.0899 mW       100.3259 mW     3.5250e+06 pW       207.3980 mW
```

- Area

```
Library(s) Used:

    tsl18fs120_typ (File: /tools/kits/tower/PDK_TS

Number of ports:                          162
Number of nets:                           152
Number of cells:                            3
Number of combinational cells:              0
Number of sequential cells:                 0
Number of macros/black boxes:               0
Number of buf/inv:                          0
Number of references:                       3

Combinational area:               87587.500000
Buf/Inv area:                     12138.000000
Noncombinational area:           103377.750000
Macro/Black Box area:                 0.000000
Net Interconnect area:            88534.872341

Total cell area:                 190965.250000
Total area:                      279500.122341
```

Timing

```
 # A fanout number of 1000 was used for high fanout net computations.
Operating Conditions: tsl18fs120_typ   Library: tsl18fs120_typ
Wire Load Model Mode: enclosed
  Startpoint: ctrl_read_mode_U1/Counter_reg[0]
              (rising edge-triggered flip-flop)
  Endpoint: CVU_OUT_bitmap[63]
            (output port)
  Path Group: (none)
  Path Type: max
  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  project            280000                tsl18fs120_typ
  mux_16_1           4000                  tsl18fs120_typ

  Point                                                    Incr       Path
  ---------------------------------------------------------------------------
  ctrl_read_mode_U1/Counter_reg[0]/CP (dfnrq1)             0.00 #     0.00 r
  ctrl_read_mode_U1/Counter_reg[0]/Q (dfnrq1)              0.65       0.65 r
  ctrl_read_mode_U1/Sel[0] (ctrl_read_mode)                0.00       0.65 r
  CVT_1_U1/ReadSel[0] (CVT_17_10)                          0.00       0.65 r
  CVT_1_U1/mux_16_1_U1/sel[0] (mux_16_1)                   0.00       0.65 r
  CVT_1_U1/mux_16_1_U1/U792/ZN (inv0d0)                    0.17       0.82 f
  CVT_1_U1/mux_16_1_U1/U791/ZN (nr02d0)                    0.43       1.25 r
  CVT_1_U1/mux_16_1_U1/U11/ZN (nd02d1)                     0.24       1.49 f
  CVT_1_U1/mux_16_1_U1/U71/ZN (inv0d1)                     0.58       2.07 r
  CVT_1_U1/mux_16_1_U1/U133/ZN (aoi22d1)                   0.11       2.18 f
  CVT_1_U1/mux_16_1_U1/U131/ZN (nd04d0)                    0.14       2.32 r
  CVT_1_U1/mux_16_1_U1/U125/Z (or02d0)                     0.12       2.44 r
  CVT_1_U1/mux_16_1_U1/mux_out[63] (mux_16_1)              0.00       2.44 r
  CVT_1_U1/ReadData[63] (CVT_17_10)                        0.00       2.44 r
  CVU_OUT_bitmap[63] (out)                                 0.00       2.44 r
  data arrival time                                                   2.44
```

7. Layout

Layout was done with Tower 0.18u Design Kit and Innovus 15.2. The process have 6 building steps:

1. Reading the technology specification files.
2. Reading the synthesized Verilog file.
3. Initiating and setting the primary Floorplan.
4. Cells placement.
5. Implementing the power supply net.
6. Wiring the design.

More about this procedure can be found under the link

http://webee.technion.ac.il/vlsi/Projects/Manuals/innovus56_tsl018.pdf

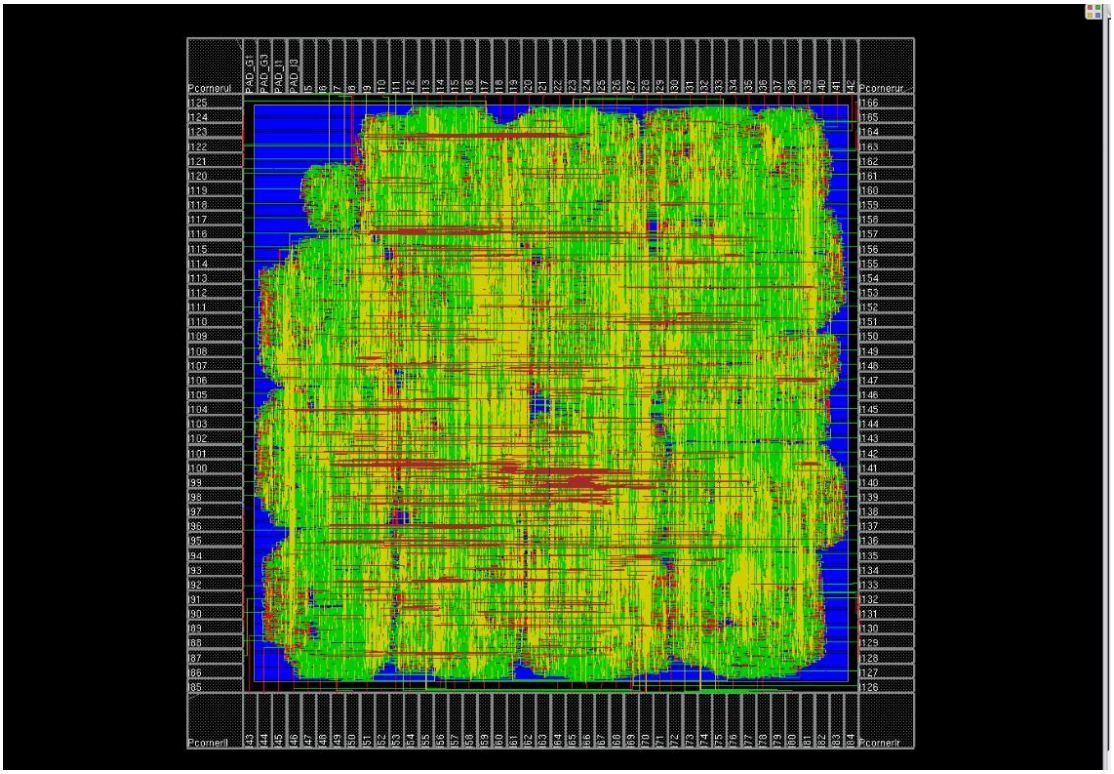


Figure 14: Layout of our design

8. Conclusions

This chapter summarizes up our experience during the work on the project and the main aspects of the design. We implemented our components according to its specifications that are described in the pages of VGIW architecture. We experienced the difficulties that can be faced during a standard design mission.

In this new project we experienced the work with old and new components and the need to write a clean code ,readable ,respects the System-Verilog specifications, and the most important part is to obey each components restrictions so integrating them to one unit would be easier , and more sophisticated .

9. References

(1) https://www.tacc.utexas.edu/documents/13601/88790/8Things.pdf
(2) http://www.c-jump.com/CIS77/CPU/VonNeumann/lecture.html#V77_0010_von_neumann
(3) https://en.wikipedia.org/wiki/Dataflow_architecture
(4) https://en.wikipedia.org/wiki/Control_flow
(5) Single-Graph Multiple Flows: Energy Efficient Design Alternative for GPGPUs
(6) Control Flow Coalescing on a Hybrid Dataflow/von Neumann GPGPU