Politecnico di Torino

III Facoltà di Ingegneria

Final project
Parallel high level synthesis using GPU

Master degree in Software Engineering

GPU programming

Dilillo Nicola S284963

GitHub Repository

Contents

CHAPTER 1

Introduction

1.1 Motivation

A lot of EDA tools used heuristic approaches to solve synthesis for electronic circuits. In high level synthesis (HLS) there could be two types of constrained: area or time. Exact algorithms are not use to solve those kinds of problems because solutions will occur after a lot of time, even years. In this project the main purpose is to exploit the parallelism of the GPU to create all the possible set of combination of resources, that satisfy area constrain, and through the usage of a list scheduling algorithm evaluate which is the fastest one.

To evaluate all the benefits of GPU algorithm a CPU version has been written. The basic concept is the same but in the latter a the classical sequential approach has been used.

1.2 Terms clarification

Before to move on a little refresh about general topic of this project is mandatory.

Synthesis is the process that generate the circuit netlist of a certain circuit model, given a set of library and a target device. High level synthesis (HLS), also known as behavioural synthesis and algorithmic synthesis, is a design process in which a high level functional description of a design is automatically compiled into a RTL implementation that meets certain user design constraints.

HLS is divided in three main phase:

- Allocation, choice of how many resources will be used;
- Scheduling, of the processes on the set of resources, assigning one of them;
- Binding of operations to specific resources.

Solution that work simultaneous on these three problems would lead to better results (closer to the Pareto-optimal set), but it is typically too complex. A Pareto point is a solution that is better than the otherwise under certain aspect and is not dominated by anyone.

1.2.1 List scheduling

List scheduling is a greedy algorithm that takes as inputs a list of jobs, ordered according a priority criteria, that is executed on a set of given resources. The algorithm repeatedly executes the following steps until a valid schedule is obtained:

- Take the first job in the list (the one with the highest priority).

- Find a resources available for executing the job.
- If a resources has been found, schedule the job otherwise select the next one from the list.
- Repeat until resources are not more available or there are not more node to schedule.

To obtain a valid schedule all nodes have to been scheduled in a finite time respected the resources number constraints.

1.2.2 Example

Let's try to synthesis the following function $F = a * b + a * c$, that is represented by the DFG in figure 1.1. It's important to note that each node of the DFG is binary, receive at most two inputs, and that each node will be assign to a specific resource in a given instant of time.

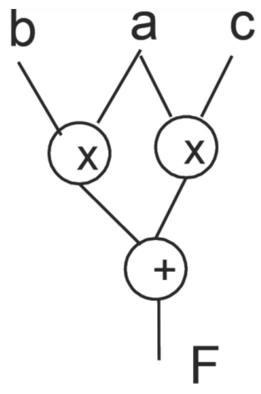


Figure 1.1: DFG of $F = a * b + a * c$

Two possible schedules could be picked, like shows in figure 1.2. Both are Pareto-optimal but in the left one just two resources have been used (one adder and one multiplier) while in the right one is present a further multiplier. Of curse the latter has a smaller latency.

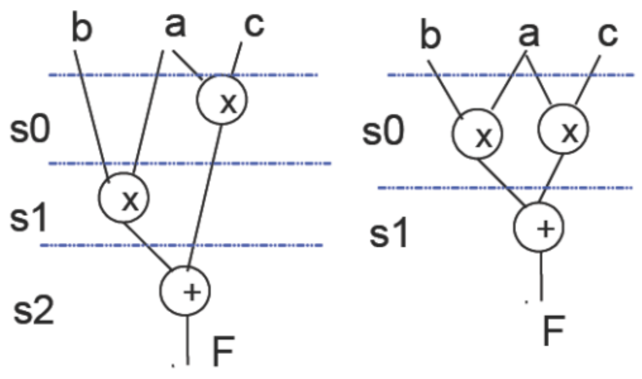


Figure 1.2: On the left there is a bigger latency but less area is used, while on the right is the opposite

The purpose of this project is to select the set of resources that allows to have the lowest latency and that respect a certain value of area constraint. In this case let's suppose that each resources has an unitary value of area and the limit is 2, the first schedules will be the one chosen.

The schedule is then convert in hardware, like shows in figure 1.3.

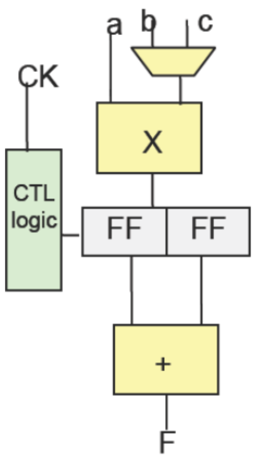


Figure 1.3: Hardware implementation

1.3 How to proceed

The synthesis tools used to schedule DFG through the usage of efficient heuristic solutions that find optimal solutions because those approaches are not exhaustive and don't use exact scheduling algorithm (like ILP).

The purpose of this project is to explore all the set of resources combinations and, based on a list scheduling algorithm, choose which one retrieve the smallest latency. In order to do that GPU is exploited to create all those combinations in parallel and and then to performer the schedule on the given set.

As first attempt a sequential version, runnable on CPU, has been written and then and different versions for GPU, in order too see which one will be the fastest.

All the available DFGs used for test have been taken from another course and for this project purpose the format has been a little modify, through a python algorithm (appendix A), to have a faster computation in cuda language. The original one are available and usable to retrieve a visual interpretation of operations.

CHAPTER 2

Parallelization

The advantage of GPU implementation derive to the possibility to create all possible combinations of resources in a parallel way and not sequentially. Theoretically this means that in a single instant the entire set is created, practically little amount of time pass between one and other and, if the set is too big, parallelization is split in more blocks.

2.1 Combination

Formula 2.1 indicates all to possible combinations that can be extract from a set of n elements choosing group of k elements. In this case the n indicate the occurrence of resources.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \tag{2.1}$$

The complete power set to formulate has an occurrence equal to the sum of all combination from the minimal number of resources needed to the max number, like show in formula 2.2.

$$\sum_{i=k_{min}}^{k_{max}} \binom{n}{i} = \sum_{i=k_{min}}^{k_{max}} \frac{n!}{i!(n-i)!} \tag{2.2}$$

2.2 Combinadic

Combinadic is a useful technique that, giving and index in the range between 0 and $\binom{n}{k} - 1$, return a unique combination set of k element.

From a given combination is possible to find the corresponding number N, corresponding to $c_k > ... > c_2 > c_1$, according formula 2.3.

$$N = \binom{c_k}{k} + ... + \binom{c_2}{2} + \binom{c_1}{1} \tag{2.3}$$

From number N is more difficult extract the combination. By the definition of the lexicographic ordering, two k-combinations that differ in their largest element c_k will be ordered according to the comparison of those largest elements, from which it follows that all combinations with a fixed value of their largest element are contiguous in the list. Moreover the smallest combination with c_k as the largest element is $\binom{c_k}{k}$, and it has $c_i = i - 1$ for all $i < k$ (for this combination all terms in the expression except $\binom{c_k}{k}$ are zero).

Therefore c_k is the largest number such that $\binom{c_k}{k} \leq N$. If $k > 1$ the remaining elements of the k-combination form the (k-1)-combination corresponding to the number $N - \binom{c_k}{k}$ in the combinatorial number system of degree k - 1, and can therefore be found by continuing in the same way for $N - \binom{c_k}{k}$ and k - 1 instead of N and k.

2.2.1 Example

Suppose one wants to determine the 5-combination at position 72. The successive values of $\binom{n}{5}$ for n = 4, 5, 6, ... are 0, 1, 6, 21, 56, 126, 252, ..., of which the largest one not exceeding 72 is 56, for n = 8. Therefore c5 = 8, and the remaining elements form the 4-combination at position 72 - 56 = 16. The successive values of $\binom{n}{4}$ for n = 3, 4, 5, ... are 0, 1, 5, 15, 35, ..., of which the largest one not exceeding 16 is 15, for n = 6, so c4 = 6. Continuing similarly to search for a 3-combination at position 16 1 15 = 1 one finds c3 = 3, which uses up the final unit. This establishes $72 = \binom{8}{5} + \binom{6}{4} + \binom{3}{3}$, and the remaining values c_i will be the maximal ones with $\binom{c_i}{i} = 0$, namely $c_i = i - 1$. Thus we have found the 5-combination {8, 6, 3, 1, 0}.

2.2.2 Combinadic in GPU

Exploit Combinadic in GPU is pretty immediate. Each thread in GPU has an unique id value that goes from zero to the number of initialized threads minus one. This id could be used to take a specific combination where each element from the set correspond to an id of a specific resource.

2.3 Repetition

What explain until now is based on combination with no repetition but a resource can be instantiate more than once!

In a sequential algorithm the max repetition constraint is implemented since the beginning in the code while here it is developed in thread differently. In this case also the CPU algorithm follow the same method in order to have a better prototype of the best parallel one.

Like will be show in the next chapter, it's not possible to create all repetitions of combinations set inside a single thread on the GPU, errors will arise due to time because GPU kernel can last for a limited period of time. To avoid this problem single thread should handle also each single repetition. In this way much more simpler thread are created with combinations that could not respect the area limit or that not cover all the operations. While in the previous version all the repetitions that respect area constraint are analysed and the useless are not created, now all ones are take into consideration.

To not explode computation a max of repetition for each resources is set.

2.4 Combinadic and repetition

To have repetition for combination given from $\binom{n}{k}$ and having a max number of repetition $max_{repetition}$, the threads that will be created are given by the formula 2.4.

$$M = \binom{n}{k} * max_{repetition}^{k} \tag{2.4}$$

The id thread can go from id zero to $M - 1$ and it is then split in two according formula 2.5 and 2.6.

$$id_1 = d/k \tag{2.5}$$

$$id_2 = d \mod k \tag{2.6}$$

From id_1 is extract the combination like show in section 2.2, while from id_2 the corresponding repetition.

Given id_2 is possible calculate the corresponding repetition of a given resources i according the following formula 2.7.

$$r_i = id_i \mod k \tag{2.7}$$

With i that goes from 0 to $k - 1$ and with $id_0 = id_2$. Between iteration id_i change according formula 2.8.

$$id_{i+1} = id_i/k \tag{2.8}$$

CHAPTER 3

Implementation

3.1 CPU

Before to proceed with the parallelization of this algorithm, a sequential version, executable on CPU, has been written. At the end the results will be the same and the only different will be the time of computation.

This is the prototype of final release, more easy to develop, especially for the final part that compare the best scheduling latency among all. Here memory optimization are not so useful for better performance, while algorithm improvement can speed up the computation.

Two versions are implementation:

- Version 1.0, create single combination that care about all repetitions, more efficient;
- Version 2.0, create single combination that care about single repetition, less efficient but more similar to what has been implemented for GPU;

3.2 GPU Data Structure

To perform the GPU elaboration data structure has been changed in order to lighten the overall process of memory copy, the real bottleneck of this approach. For this reason the device has to reduce the data before to pass them to host.

The main big variables are all related about node and operation. Both keep track of information that are not useful for scheduling computation and so, before to call GPU kernel, they are filtered to maintain only the main data usable for computation, the useless ones will be always available on the host.

To improve the elaboration also other size variable are been accurately choose to use as less memory as possible.

3.3 Scheduling Algorithm

The section 1.2.1 explains which algorithm has been choose to performer the scheduling operation. This part is identical both for GPU and CPU versions, change only the implementation of some variables used, that are always the same. How will be explained in the next chapter those variables, to be more precise those arrays, are one of critical points that have been improved to achieve better performance.

3.4 GPU Kernel

Different version has been created to see which one executes the fastest computation.

By the way all of them use the same mechanism to analysed which repeated combination has the best latency. In the same block all threads save the result of latency and their own repeated combination on shared memory so, when all schedules terminates, the best set of resources is picked from the same block and pass to host trough the main memory copy operation. At the end CPU checks all results coming from kernel and choose the best one with minimum latency, in case of equality the one with minimum area is chosen. This part is implemented almost in the same way in all versions.

3.4.1 Version 1

This is the basic version where each thread handle all repetition, given a combination. If not all operations are covered the thread doesn't create the repetition and goes on. The scheduling of single repetition is executed only if the area constraint is respected.

Combinations are group 1024 at a time, each using a single block inside a stream, using a max number of stream. At the same time it is possible to have block that work with different value of k ($\binom{n}{k}$) thanks to the stream.

All the arrays are allocated on shared memory in a dynamic way, respecting the max amount of shared memory for block.

For long repetitions time problems occurs and the kernel stops its execution.

3.4.2 Version 2

Each thread handle a single repeat combination and, also if the combination don't cover all the needed operation or if area constraint are not respected, the thread is created but the scheduling is not lunch. Different versions have been created to test which is the best way to arrange variable declaration.

- Version 2.0, like the previous one but with the new property of single repetition for thread.
- Version 2.1, use less shared memory that is compensated by dynamic allocation when arrays are used only inside a single block.
- Version 2.2, not use anymore dynamic allocation and improve the usage of resources, allocating space only for the ones used in scheduling.
- Version 2.3, instead of shared memory, arrays of fixed dimension are used inside each thread when possible and the global memory is called only when latency value obtain is better than previous one. Furthermore now, at the end execution of each blocks inside a thread, only thread with id zero take care to choose which has the best latency.

In al the versions the variables that take information about nodes, operations and best latency thread are keep on shared memory to allow start thread to keep the best result and the end.

From this versions is pretty evident that dynamic allocation is a technique that needed a lot of time and not the best choice while, the usage of local register of version 2.3 and call global memory as lest as possible, are big improvement.

Also here combinations are group 1024 at a time, each using a single block inside a stream.

3.4.3 Version 3

Based on version 2.3, in previous version each stream handles single block while in this version streams are not more used and each kernel call works with all blocks that, given the combination $\binom{n}{k}$, has the same value of k. Now much more global memory is instantiate at the same time and this could result in an import speed up of the system that call global memory copy only and organize the overall workload in a better way.

3.4.4 Version 4

Merge the advantage of multi streams of version 2.3 and with the possibility to work with more blocks inside each stream of version 3. Then a further improved of version 4.1 has been add in order to have the possibility to have workload that stress the GPU with the creation of billion of threads. Furthermore, the number of heave computation operations has been reduced as much as possible.

CHAPTER 4

Results

All tests, for CPU and GPU, had been run on the same device, Jetson Nano, the one provided from the course, and all results obtained are related to this hardware.

4.1 Execution

Before to run each execution be sure to compile using the available *Makefile*.

To start execution it is possible run one of the bash scripts available.

- *run.sh*, to run all the GPU version.
- *run_CPU.sh*, to run all the CPU versions.
- *run_long.sh*, to run a long execution of a particular DFG using the version 4.1, the only that can carry such workload.

To run a single version directly three element are needed.

- A file containing the DFG information, this file are obtained through original DFG file saved in .dot format. The conversion is achieved through a python script explained in appendix A.
- The value of area constrained.
- A file containing the all possible resources available. It is organized in the following way:
  - first row is the number of operations k;
  - then there are k rows containing the pair operation number and number n, representing the different resources;
  - after each of these rows there are n row containing the pair area and speed of each resource that can execute that operation.

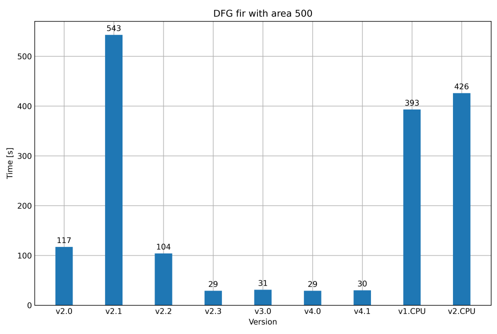
The executable can be lunched in the following way:

```
dfg.o <dfg_name>.txt <RTL_file>.txt <area_constrained>
```

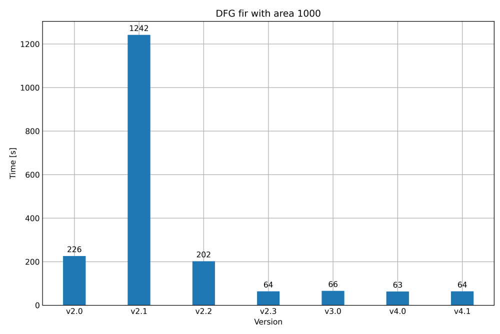
All latency, according DFG, are available inside a log file.

4.2 Comparison

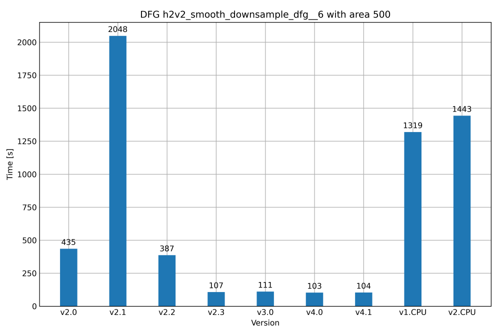
In the following graphs, obtained from python script *elaborate_data.py* (appendix B), it is possible to observe the performances of the all versions using different DFG and area constraint. A lot of other tests had been run to check not only performances but also the correctness of the scheduling but the ones show here are the most significant. It's important to notice that version 1 is useless because, when it works, all versions, GPU and CPU, have time execution almost equal to zero second.



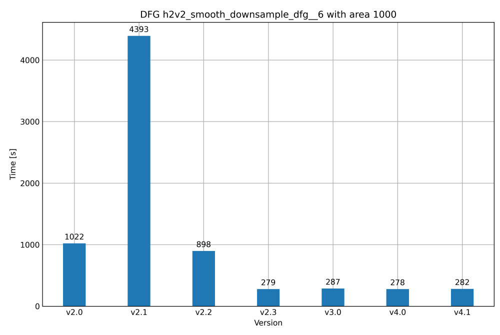
(a) fir bar graph with area 500



(b) fir bar graph with area 1000



(a) h2v2 with area 500 bar graph



(b) h2v2 with area 1000 bar graph

In the figure 4.1a and 4.2a it's possible to appreciate the advantage to exploit GPU. Beside the version 2.1 spent the greatest time, all the others have significant improvement in time, also respect to the first basic version 2.0 and 2.3.

Version 2.3, 3.0 and 4.0 have almost the same result, also if 4.0 produces always the best ones. This could be due to the virtualization process employed from GPU that, on Jetson Nano, it's quite limited or due to the Operating System internal scheduling.

More precise data are available in repository, under *result* folder.

4.3 Conclusion

The code running on GPU respect to the CPU is speed up of 14 times, that is an important quantity.

Like premeditated in chapter 3 then last version is the best one. Using the internal registers inside GPU, the fastest memory type, and using much more blocks inside each stream, has been possible the reach the best performance.

Streams have been important:

- they allowed to have a further grade of parallelization of the workload;

- they allowed to use asynchronous copy of global memory, much faster than synchronous version, of course this operation have been limitated as much as possible like explained in the above chapters.

The final version take care of all possible workload problems is the 4.1 that, cause to this little overhead, is a little heavier than the 4.0. All the other final version, like 2.3 and 3.0, don't look to have a big differences in percentage improvement but, on long execution time, important speed up could be achieved in 4.0 and the versions 4.1 could be used to handle huge quantity of threads.

APPENDIX A

Convert DFG

The extension .dot it's a file format suitable for representing DFG in a graphical way. For the purpose of this project this kind of format is not very efficient and so the following python script has been written to convert it in a more easy one. Start from the original file the script read all the node and than in the new file are written:

- the number of node n;
- n rows representing the pair name node and operation node.

Then also the edge are read and in the file are written:

- the number of edge e;
- e rows representing the pair starting node and arrival node.

The produced file is saved in .txt format.

In this way the reading in C language is much more easier and more faster and the DFG remains the same.

To se the graphical representation of DFG the .dot file is needed while to run the GPU scheduling the conversion format is used.

DFG with .dot extension are available under folder *DFGs* while DFG converted are available under folder *DFGs_new*.

APPENDIX B

Elaborate Results

When the bash scripts are run to automatize the workload, for each version, different log files are created. Inside them are present all the information about each single execution and main ones are:

- DFG name,
- Area constrain used,
- Latency obtained,
- elapsed time.

Thorough the python script *elaborate_result.py*, all these data are elaborate to create different graphs.

The bar graph is created starting from the DFG used and the area constrain applied. Then for each version is displayed the time spend to conclude the execution and, when is present, also CPU information executions are showed.