An up to date Mapping Methodology for GPUs

Florian Gouin, Corinne Ancourt, Christophe Guettier

▶ **To cite this version:**

Florian Gouin, Corinne Ancourt, Christophe Guettier. An up to date Mapping Methodology for GPUs. 20th Workshop on Compilers for Parallel Computing (CPC 2018), Apr 2018, Dublin, Ireland. hal-01759238

HAL Id: hal-01759238

https://hal-mines-paristech.archives-ouvertes.fr/hal-01759238

Submitted on 5 Apr 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An up to date Mapping Methodology for GPUs

Florian Gouin*†, Corinne Ancourt* and Christophe Guettier†
* MINES ParisTech - PSL Research University,
Centre de Recherche en Informatique, Fontainebleau, France
† SAFRAN Group, Massy, France

CPC2018 Dublin, Ireland

***Abstract*—In this paper, we propose an up to date mapping methodology for Graphics Processing Unit (GPU) architectures. The first objective is to analyse and transform the application in order to highlight the minimum parallelism necessary to fit the GPU characteristics. The second is to optimize the mapping of the application onto the GPU by taking into account all its powerful components.**

***Index Terms*—Parallel programming, Parallel processing, High Performance Computing (HPC), GPGPU, image processing.**

I. Introduction

A lot of work has been done by the compiler community on GPU mapping over the last decade. This valuable legacy can be divided in four categories: Domain Specific Language (DSL), transformation directives, skeletons and automated transformation compilers.

The *DSL* approach is a solution which hides complex architectural constraints behind a high-level specialised language. This approach implies rewriting the concerned code in its entirety to map it on GPU. The *transformation directives* are annotated directly inside the source code by using *pragmas* as for OpenACC [1] or written in a script file as for CUDA-CHiLL [8]. In a way, this approach can be considered as an extension from the DSL one by using a specific language but here in order to characterise the desired code transformations. *Skeletons* are regular code patterns whose implementation has been optimised for a given architecture. The hardest point with this approach is to select the best representative pattern for each part of code involved. Finally, *automated transformation compilers* analyse a source code and automatically apply transformations to it. Concerned parts of code are eventually mapped on the GPU.

Our objective is to manage sequential algorithm transformations to best fit on today's GPU with minimal human involvement. In a consequence, this one belongs to the *automated transformation compilers* category. As far as we know, the main representative solutions are: C-to-CUDA [4], PPCG [10], R-Stream [7] or PIPS/Par4All [2], [3].

Our methodology offers the opportunity to consider the latest GPU architectural refinements such as: *Texture* and *Surface* memory usage, asynchronous communications or kernel concurrency. It also considers state of the art optimisations like reducing memory transfers between the host processor and the accelerator(s) or using on-chip high-speed memory. Our approach has been evaluated with NVIDIA GPUs but it can be easily extended to other GPUs as the AMD's one.

Our methodology is built on five stages. The first performs static and dynamic program analyses. Data dependencies are identified and the global application is represented by an Abstract Syntaxic Tree (AST) using interprocedural analysis. The second stage performs code and loop transformations to improve the code affinity with the GPU architectural constraints. Appropriate application criteria for the transformations have been defined. The third stage is kernel optimisation. GPU kernels are modified by tuning loop granularity to improve runtime performance. The fourth stage applies some code specialisations by using the GPU specific refinements described above. Finally, in the fifth stage, GPU codes are generated for the host processor and the accelerator(s).

We evaluated this methodology on an industrial application case. In this way, the *simpleflow* algorithm [9] has been used to evaluate the quality of the mapping generated by our methodology. This algorithm is available in the contribution repository of the well-known image processing library, OpenCV. A global speed-up of 13 is obtained during the second step by mapping the sequential program from an ARM Cortex A57 quad-cores Central Processing Unit (CPU) to an Nvidia Maxwell GM20B 256 cores GPU architecture.

II. Mapping methodology

The global mapping methodology is illustrated in figure 1. Five main steps are represented to efficiently map a sequential algorithm on heterogeneous architecture composed of a main CPU processor[1] and one or many GPUs. Firstly, the original source code is analysed by using static and dynamic approaches. This point is developped in section II-A. Secondly, loop nests identified during the previous step are categorised in section II-B by separating GPU compatible ones from those which will remain on CPU. The next step concerns mapping optimisations. Because this step remains optional in the methodology, its details are omitted here. The fourth step considers architecture specialisations in the dedicated section III. Finally, the methodology comes to an end in section II-C with the code generation process.

[1] Usually called the *host processor*

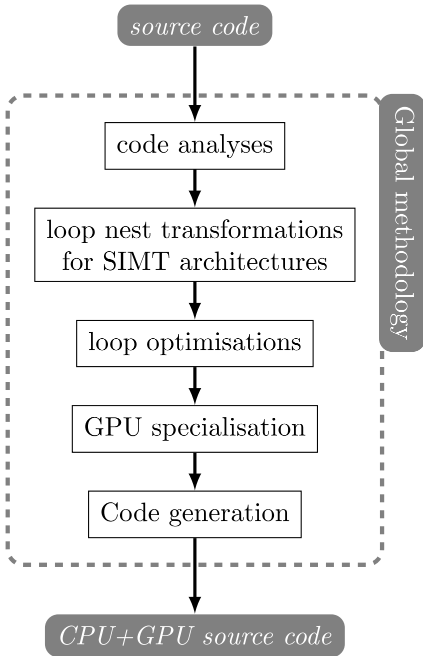

Figure 1. Global GPU mapping methodology

A. Code analyses

The code analysis in our methodology is composed of two parallel analyses. On one hand, the static code analysis is used to extract the code specifications. On the other hand, a dynamic code analysis is used to extract algorithm metrics.

The ***static code analysis*** illustrated in figure 2, starts with the detection of four elements inside the source code: loops, function calls, array accesses and branches. All remaining parts of code not concerned by the previous elements are considered as basic blocks. The *loops* represent a potential source of parallelism and might be mapped on GPU by using internal schedulers. Due to the interprocedural approach of our methodology, *function calls* are tracked inside the original code. Moreover, the *mono-procedural* operation of GPU kernels requires to remove all nested function calls. *Array accesses* represent possible space memory communications and are considered as potential source of data parallelism. Finally, the *branches* can have a major impact on dependencies and so in parallelism. Moreover, at the finest grain parallelism, branches can lead to a degradation of streaming multiprocessor performances, explained by their vectorial conception. That is why we pay particular attention to branches.

After this first collecting step, we use the *dependance analysis* to determine loop parallelism. In this way, flow dependences, anti-dependances, output and input dependences are computed for scalar and array type variables. Scalar privatization will be applied on parallel loops. Array elements are addressed by analysing the image of the *loop iteration* domains under the *array access* functions and the convex array regions [5].

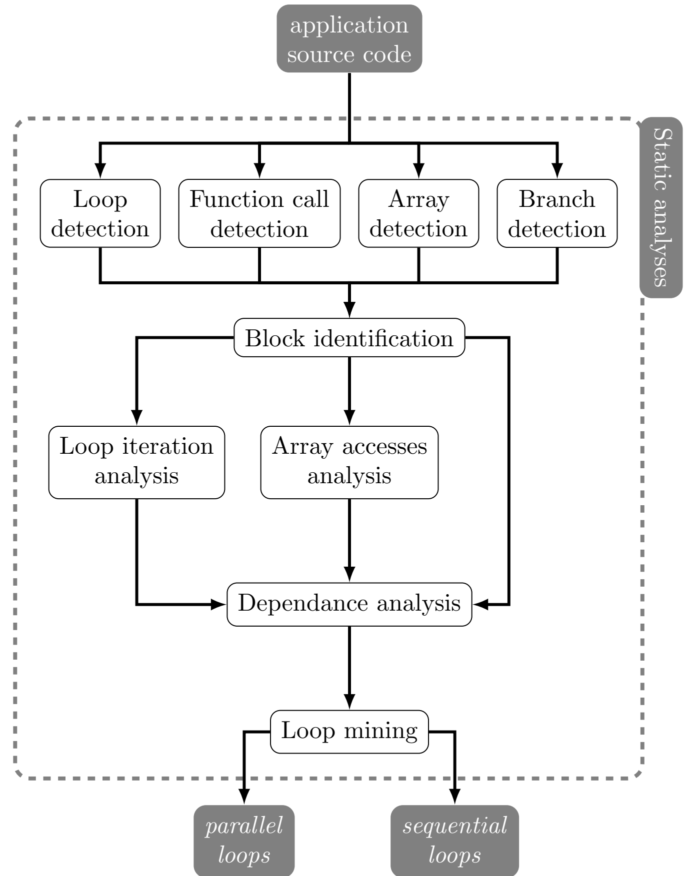

Figure 2. Static code analysis phase

During the *loop mining* step, loops are finally categorised. The ones concerned by embedded dependencies are considered sequential and the others parallel.

At the end of this static analysis, we have all the required elements to generate an intermediate representation of the original code. Our intermediate representation is based on an AST coupled with a dataflow representation.

The ***dynamic code analysis*** from figure 3 is used to determine runtime metrics. Here, we consider the global application runtime, loop and function runtimes. Those metrics are used later in the methodology to validate the loop nest mappings on GPU. Loop runtimes are used to validate GPU kernels while function runtimes are used to validate communications between the CPU and the GPU. The global application runtime is used to appreciate the whole GPU mappings speedup. By using these metrics, we prevent any accidental devaluation of the application runtime.

B. Loop mapping methodology

During the previous static code analysis, we have distinguished parallel loops from sequential ones. Now, we need to identify loop nests that match with GPU architecture constraints. This verification task is done by the ***GPU loop identification*** process. It is composed of three criteria visible in figure 5. Each loop nest compatible with those three criteria can be mapped on GPU.

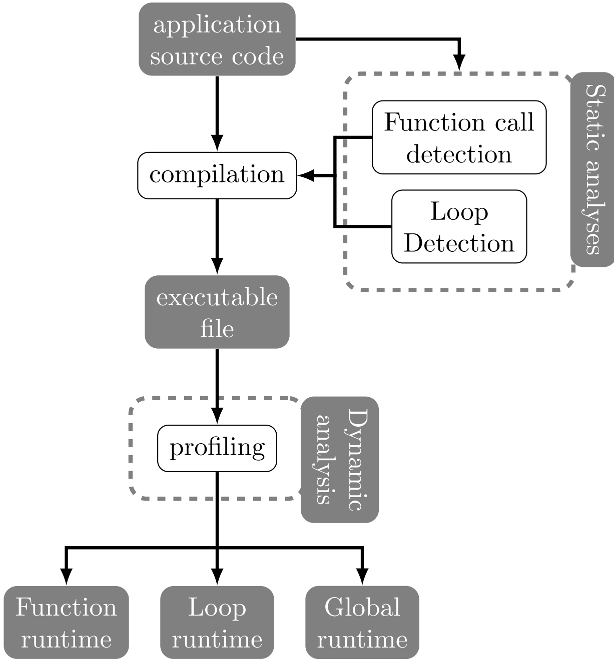

Figure 3. Dynamic code analysis phase

The **loop nest pattern** is the first of these criteria. It verifies the shape of each loop nest and extracts the GPU compatible parts as described for Nvidia architectures in figure 4. Up to six nested loops can be managed by the GPU. These nested loops have to be divided in two sets of three loops maximum each corresponding to *block loops* and *thread loops*. The former has the specificity to be imperatively parallel but the latter can embed a dependance and thereby be sequential or parallel. This segmentation is relative to the two–level computing hierarchy of the GPUs which is made up of *cuda cores* encapsulated in *streaming multiprocessors*. In consequence, the *block* loops will be mapped on *streaming multiprocessors* and the *thread* ones on *cuda cores*. The block loops mapping criterion is described in the formula 1 and the thread loops in formula 2.

$$b_n \Rightarrow \exists n \in \mathbb{N},\ 0 < n \leqslant 3 \tag{1}$$

$$t_p \Rightarrow \exists p \in \mathbb{N},\ 0 \leqslant p \leqslant 3 \tag{2}$$

The **loop nest iteration size** is the second criterion. GPUs can generate a limited amount of iterations for each mapped loop. In consequence, the purpose of this criterion is to make sure that the potential GPU loop iterations remain lower than the architectural limitations. An example for Nvidia architecture is given in the formula 3. All the parameters are strict constraints except for the ones written in orange which correspond to mapping

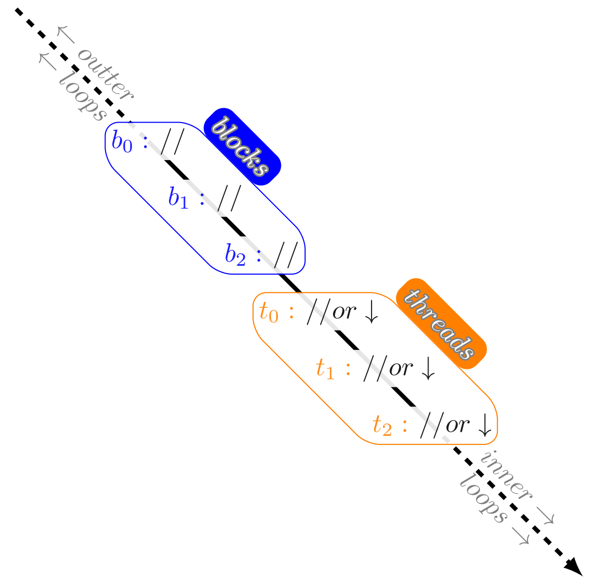

Figure 4. Criterion 1: GPU loop nest pattern

optimisation parameters.

$$\begin{cases} b = b_0 \times b_1 \times b_2 \\ b_0 < 2147483647 \\ b_1 < 65535 \\ b_2 < 65535 \\ b \gg t \end{cases} \quad \begin{cases} t = t_0 \times t_1 \times t_2 \\ t < 1024 \\ t_0 < 1024 \\ t_1 < 1024 \\ t_2 < 64 \\ t\%32 = 0 \\ t > 4 \times 32 \end{cases} \tag{3}$$

The **loop nest memory size** is the latest of these criteria. The memory footprint of the selected loop nest is computed by using the memory accesses and the loop iterations. Both of them have been determined during the static or dynamic analysis phase. If the memory footprint is bigger than the available GPU global memory, the loop nest could not be mapped on GPU. This criterion is described in the formula 4.

$$Global\ memory_{footprint} < GPU_{memory} \tag{4}$$

The ***loop transformation*** process is an important part of the mapping methodology. Previously described criteria represent strict constraints. In consequence some loop nests are rejected whereas they could become GPU candidates after some legal code transformations driven by the dependance analysis. So, we have selected a set of loop transformations to be applied in order to increase the quantity of potential loop mapping on GPU. This set is summarized on the right side of the figure 5. Ticked loop transformation patterns match the ones improving GPU mapping for the corresponding criterion.

For the **loop pattern** criterion, *loop fusion* and *fission* are used to transform imperfect nested loops to perfect ones as represented in the figure 2. In other words, they transform loop nest composed of multiple loops with same

depth to one[2] or many loop nests[3] with a single loop per depth level. The rest of the perfecting loop nest work is done later in the methodology during the *inter loops block motion* phase. The *loop coalescing* and *tiling* have an impact on the depth of the loop nest. The former increases the number of loops mapped on GPU by compacting the loop nest depth. The latter has an opposite effect. These two transformations also have an impact on the number of iterations for concerned loops. In consequence, they have to be used in correlation with the *loop size criterion.* The *parallel loop reduction* and the *tiling* can transform some sequential loops into parallel ones by reordering embedded dependances. Finally, the *loop inter change* also called *loop swapping* modifies loop positions inside the loop nest. This last transformation is particularly useful to shift sequential loops inside *thread loops* or outside the GPU mapped loops.

Concerning the **loop size criterion**, *loop tiling*, *strip mining*, and *splitting* commonly modify the quantity of loop iterations. This quantity has to be lowered to fit in criteria established in the formula 3.

Finally, the **memory size** criterion is managed by using *loop fission*, *tiling* or *strip mining.* They all can have an impact on reducing the memory footprint.

C. Code generation

Code generation steps are represented in the figure 6. The **inter GPU loops block motion** transforms each imperfect nested loop in a perfect one. This process is complementary to the *loop fusion/fission* transformations presented in the previous section. Here, the purpose of this task is to move remaining blocks of code located between the consecutive GPU qualified loops.

Next, a **space iteration densification** is used. GPU generates thread and block identifiers starting from zero to n with an unit stride. In consequence, unnormalized loops have to be transformed to fit with GPU iterations.

Because Cuda kernel cannot invoke function calls except for dynamic parallelism, **function inlining** is used to generate a mono–procedural GPU function.

Then, **kernel function outlining** is applied on GPU loop nests to generate kernel function headers and necessary communication between host and accelerator.

The final step of the methodology is about **code generation**. During this process, the intermediate representation of GPU loop nests is transformed in Cuda source code. Moreover, communication instructions between the host and the accelerator are generated on the host side at the same time. Finally, rejected loops are maintained on CPU. In consequence, the corresponding source code remains unchanged.

III. Specialisation methodology

This section focuses on the **memory specialisation** branch of the methodology. It only concerns the left part of the figure 7. The two other parts concern the concurrency specialisation and the communication specialisation. In the first, we consider multi-GPUs mapping as well as the usage of the height cuda instructions pipelines running in concurrency on Nvidia architectures. In the second, we identify and generate communications between the host processor and the accelerator. Moreover, a communication optimisation step has been added to remove redundant communications between kernels. Finally the communication placement can manage synchonous and asynchronous memory communications.

The GPU architecture is a multi–level memory space. This conception almost explains[4] the faster memory throughput of GPUs. The memory hierarchy is composed of a global memory available for all threads and a local memory. This latter is in reality part of the global memory except that it has the same lifetime as its referring threads. Local and global memories are the slowest inside the GPU. This hierarchy is mono–dimensional. That is why we have added an array access linearisation step in the methodology. On the opposite, registers are the fastest memory as for most of architectures. In return, the register space is small and limited for each multi-processor. Last, the register lifetime is marked out by its referring thread existence. Between them are the *texture and surface memories* , the *constant memory* and the *shared memory* addressed in the respective sections III-A, III-B and III-C.

A. Texture/Surface memory usage

Texture and surface memories are part of the global memory and in consequence inherit all its characteristics. However, they use a dedicated texture cache optimized for data locality in the case of two dimensional array accesses. Moreover, hardware elements for those memories are dedicated to manage image boundaries and linear interpolation for adjacent elements in the array. These functionalities are particularly useful for image processing applications and more specifically for convolution applications.

The distinction between texture and surface memories usage is guided by the array region analysis [5]. In the case where all accesses are read type, the texture memory could be used. On the contrary, if they are some write accesses on the array, the surface memory has to be used.

Finally, to determine if the texture and surface memory usage could lead to a speedup is not trivial. In concrete terms, we have to recognize an interpolation access form, a memory boundary management pattern or a two dimensional locality access. Analysing image of the *loop iteration* domains under the *array access* functions is the key element.

B. Constant memory usage

As for texture and surface memories, constant memory is also part of the global memory. Again, it uses a specialized constant cache. But this one does not consider

[2]In case of loop fusion
[3]In case of loop fission
[4]Memory coalescing inside *blocks* is another main reason

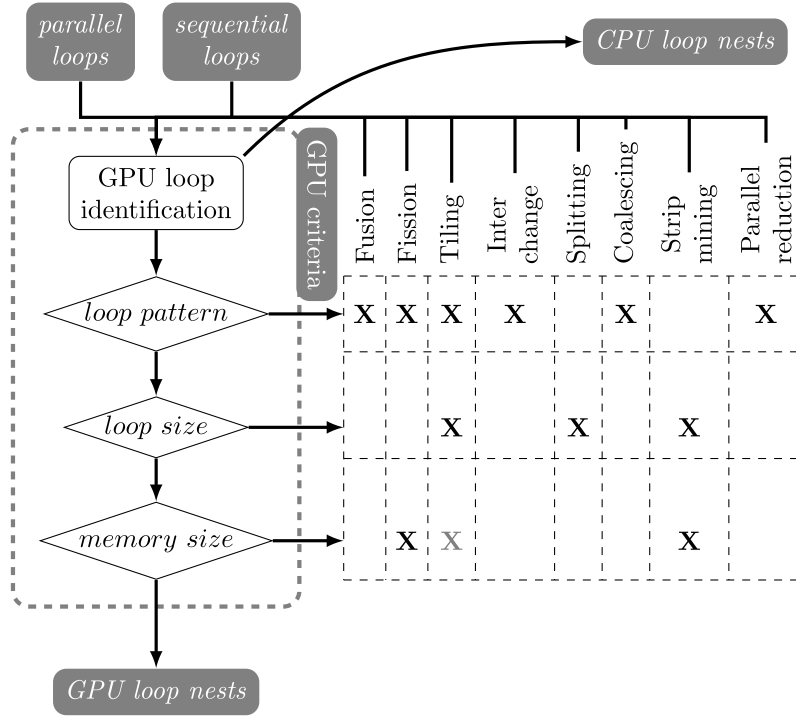

Figure 5. GPU criteria and loop transformations

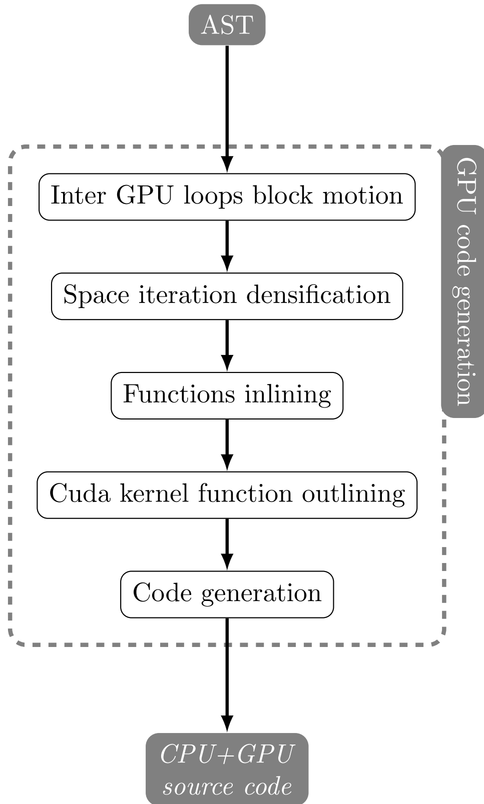

Figure 6. Code generation phase

any cache coherency. This behaviour can be explained by the usage of read-only memory space. So, the array accesses analysis is used here to determine the kind of memory accesses. If all accesses are read type, the constant memory could be used. However, the size of the constant memory is globally very limited. In consequence the memory footprint criterion will be used additionally on the corresponding loops to determine if the usage of this memory is possible. Last, lifetime of this memory is correlated to the GPU context existence.

C. Shared memory usage

The *shared memory* is located inside each multiprocessor. It has the same lifetime as its referring block of threads. In consequence, this memory can be shared by all the threads included inside the same block. The usefulness of the shared memory utilisation is defined by using jointly on the thread loops, the array access and the loop iteration analyses along with the computed dependances. If the array accesses and the loop thread iterations reveal any array element reuse, the shared memory usage presents an assured advantage on the loop runtime. If the computed dependences show embedded dependencies linked to the array accesses, the shared memory can be legally used by employing Cuda synchronisation instructions.

Shared memory presents throughput performances close to register ones. But as for the latter, this memory suffers from a very limited size. Again, the memory footprint criterion is used here to demonstrate the usage feasibility.

IV. Results

As a bench case, we have applied this methodology on the *simpleflow* algorithm [9]. We have used a Nvidia Tegra X1 platform which is an embedded GPU architecture, to establish the benchmarks.

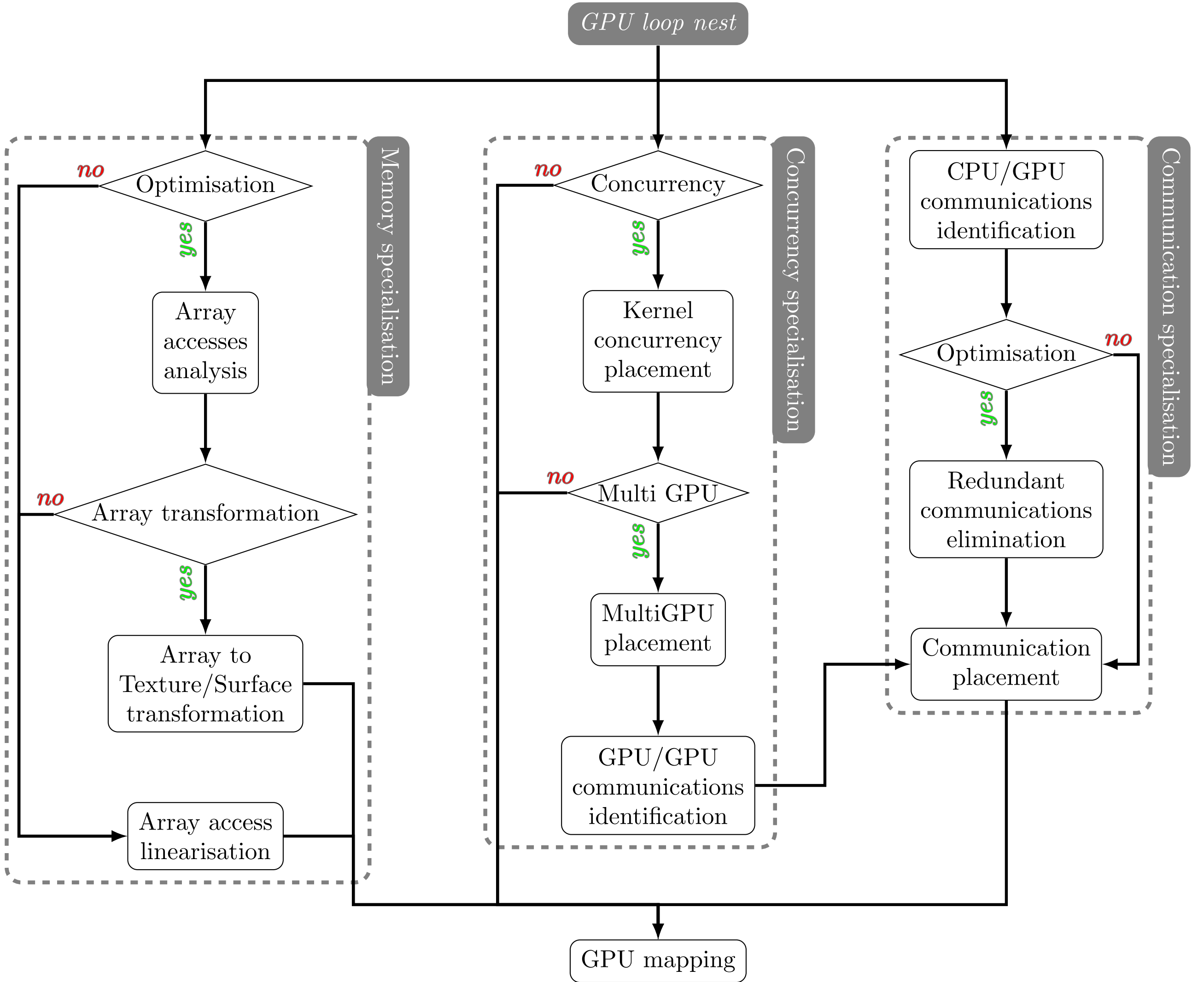

Figure 7. Architecture specialisation phase

Runtime benchmarks are compiled in a circular diagram. The execution time progresses in anti–clockwise. Grey parts of the diagram are runtimes of functions, blue ones of CPU mapped loop nests and green ones of GPU mapped loop nests. Algorithm depth is numbered from zero to five: zero being the root function and five the innermost function or loop nest.

Runtimes of the original algorithm are represented in the figure 8. This algorithm is a sequential one which uses a single core of the four available on the ARM CPU. In the figure 9 we can observe the impact on runtimes after having applied the presented parts of our mapping methodology on the original algorithm. As a result, the global application is affected by a speedup of thirteen. Moreover, we can notice that most of the loops have been mapped on GPU.

V. Further work

The methodology covered in this article refers to a loop mapping methodology added by an architectural specialisation phase. The mapping optimisation part is addressed in detailed in the report [6]. This one increases the speedup presented in this article. We are currently working on an innovative dynamic approach to improve the initial mapping on GPU. This work would come out as a better usage of GPU architecture capacities.

Finally incorporating our methodology in an automated transformation compiler will be the ultimate step.

VI. Conclusion

We have described in this article some of the most important parts of our GPU mapping methodology. The loop mapping methodology part enables to map a sequential algorithm on GPU whereas the specialisation part improves the usage of the GPU architecture capacities. In its current state, our methdology achieves an average speedup of thirteen on a complex image processing bench application. This speedup has been obtained by using a Nvidia Jetson TX1 which is a low power integrated CPU and GPU platform.

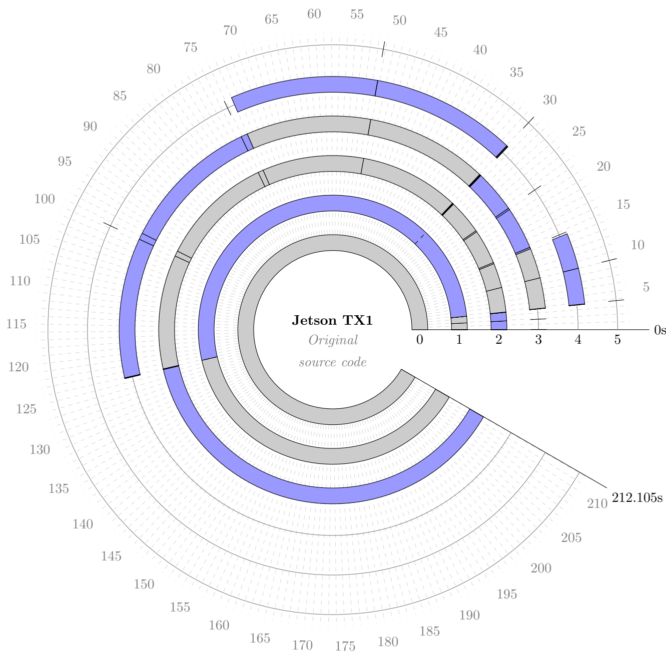

Figure 8. Original Simpleflow algorithm execution on TX1 GPU

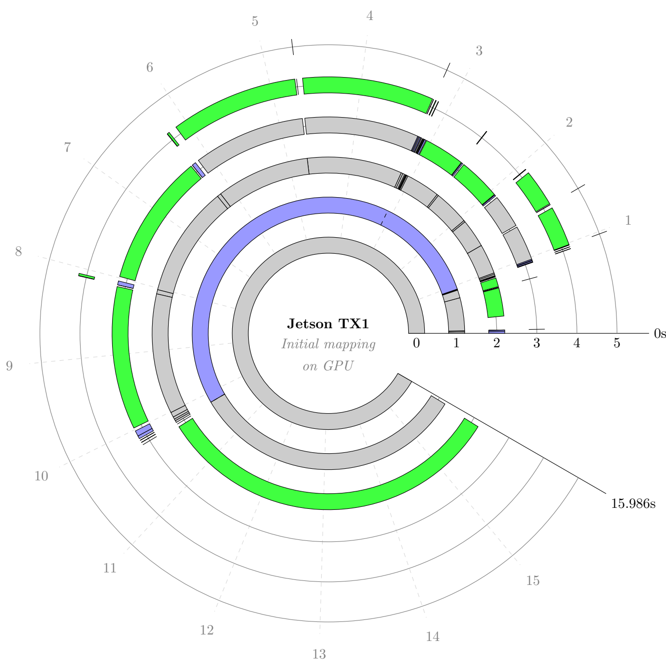

Figure 9. Simpleflow algorithm initial mapping on TX1 GPU

References

[1] OpenACC. http://www.openacc.org.

[2] Mehdi Amini. *Source-to-Source Automatic Program Transformations for GPU-like Hardware Accelerators*. PhD Thesis, Ecole Nationale Supérieure des Mines de Paris, December 2012.

[3] Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, Serge Guelton, François Irigoin, Pierre Jouvelot, Ronan Keryell, and Pierre Villalon. PIPS is not (just) polyhedral software adding GPU code generation in PIPS. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011) in conjonction with CGO 2011*, 2011.

[4] Muthu Baskaran, J Ramanujam, and P Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Compiler Construction*, pages 244–263. Springer, 2010.

[5] Béatrice Creusillet and François Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming*, 24(6):513–546, 1996.

[6] Florian Gouin. Performance optimization and profiling of image processing algorithms on parallel architectures, 2018.

[7] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. A mapping path for multi-gpgpu accelerated computers from a portable high level programming abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 51–61. ACM, 2010.

[8] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. A programming language interface to describe transformations and code generation. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 136–150. Springer, 2010.

[9] Michael W. Tao, Jiamin Bai, Pushmeet Kohli, and Sylvain Paris. Simpleflow: A non-iterative, sublinear optical flow algorithm. *Computer Graphics Forum (Eurographics 2012)*, 31(2), May 2012.

[10] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.