Automatic Code Generation for SIMD Hardware Accelerators

Serge Guelton[1], François Irigoin[2], and Ronan Keryell[3]

[1] Télécom Bretagne, HPCAS, France, serge.guelton@telecom-bretagne.eu
[2] Mines-Paristech, CRI, France, irigoin@cri.mines-paristech.fr
[3] HPC Project, France, rk@hpc-project.com

Abstract. SIMD hardware accelerators offer an alternative to many-cores when energy consumption and performance are critical. For scientific computing, GPGPUs are used in many computers of the top-500. But embedded processors also use accelerators.
However such heterogeneous platforms trade ease of developments for performance: The application code and the data must be split between the host and the accelerator, synchronizations and communications between host and accelerator must be added, and accelerator hardware constraints must be taken into account by the programmer.
To ease application development, we present an algorithm to automatically externalize the execution of a parallel loop using a synchronous master/slave protocol.
The source-to-source transformation process is incrementally consistent and the transformed code can be executed at any step of the transformation algorithm, which make application and compiler debugging easier and possible without the target hardware.
This article details the source-to-source parallel loop externalization algorithm and the new elementary program transformations which are used. Unlike previous work based on annotations or polyhedral model, it relies on interprocedural array region analysis, which embraces more situations than the polyhedral model while still not relying on user input.
It has been implemented in the PIPS framework and its results are given at each transformation step for an image processing example.

Keywords: compilation, source-to-source transformation, convex array regions, heterogeneous computing, SIMD architecture

1 Motivation and Context

Ranging from General Purpose Graphical Processing Unit (GPGPU) to Field Programmable Gate Array (FPGA), hardware accelerators are widely used to deliver high performance for specific applications. To benefit from the promised speedups, developers have to write machine-level code (VHDL, assembly ...), to map their algorithms to complex video primitives (OpenGL) or to master dedicated extensions (DSP libraries).

Numerous attempts have been made to enhance the programmability of such accelerators by using `C` as a base language. Most of them[18,10,9] rely on extensions to match hardware specificities. However, portability is lost and the generation process is a black-box for the developer, who does not understand the mapping between the code and the generated program. Moreover, except for CUDA, such an approach does support code debugging and provides very little way of the simulating resulting code. Finally, there is an obvious duplication of efforts, for hardware accelerators share some high-level concepts, while being very different in their implementation.

Offloading computations on a remote accelerators with its own memory is similar to improving cache efficiency [7,14]: Code must be reorganized in order to improve locality, and loops be sized for live data to fit into the cache [6,3]. Data layouts can be modified. In both cases, we want to minimize the memory traffic.

Code generation for hardware accelerators has regained in popularity since the appearance of GPGPUs, but it often relies on user-provided informations through annotations [15,11,5,19] and target only a kind of accelerator.

The paper by Leung & al. [16] presents a sequence of program transformations to map so-called "text-book C" code onto a GPU using the R-Stream compiler. Although R-Stream is based on the polyhedral model, the mapping process is based on phases, with decisions made early more or less outdone by later phases (e.g. loop fusion after the scheduling phase). The general R-Stream framework dedicated to homogeneous computing is re-used and key issues such as communication generation are left out of the paper. Also, array allocation and layout are dealt with late in the optimization process.

Unlike their approach, we address heterogeneous computing head first. We generate code both for the host and the accelerator. We optimize array allocation on the accelerator globally and we generate communication code. We also add new iterations, which is not part of the polyhedral model. Furthermore, our user keeps control of the code transformation at each step: source code is available to understand and to execute. However, we do not deal with all GPU specifics as our primary target accelerator is a custom SIMD processor.

In this paper, we propose a study of concepts and constraints shared by hardware accelerators and propose generic, high-level transformations that can be parameterized to fit their specific needs. Those transformations rely on interprocedural array region analysis for which we introduce concepts in §3.

We finally focus on a FPGA-based specific accelerator dedicated to image processing that embodies most of the studied issues. On this practical example and a challenging code that requires interpocedural analysis of array of structures, we demonstrate the validity of the approach by exhibiting compilation steps.

All experiments are carried out on the PIPS [13] source-to-source compiler infrastructure.

2 Context

At first glance, there is not much in common between a GPGPU, the SSE instruction set and FPGA-based machines. One is accessed *via* PCI bus to perform high performance computing, the other is located on the CPU and traditionally enhances multimedia computations and the last may be an embedded hardware used for videos surveillance. But experience shows that while very different, hardware simd accelerators share some very similar high-level concepts and impose one kind of constraints to the developers[20].

2.1 Execution Model

They all share the same execution model:

1. Some data are copied-in (a/synchronously) to a remote accelerator, copies which often suffer from a limited bandwidth. Depending on the target, this implies an allocation process or a manual handling of accelerator memory;
2. a *computation kernel* is activated;
3. data resulting from the computation are retrieved by the host.

This is the traditional *Load - Work - Store* model. GPGPU's and SSE obviously use this model.

2.2 Memory Constraints

As a consequence of this execution model, memory transfers play a critical role for performance and must be dealt with very carefully. We can observe the following similarities:

data transfers overhead: we must limit them to a minimum;
remote memory is limited: computations may have to be split, raising various issues at the boundaries;
data transfer are constrained: data alignment may be required or preferable, and Direct Memory Access (DMA) may be capable of transferring only fixed size amount of data (think of the SSE `_mm_load_ps` instruction).

As an illustration, the Terapix simd accelerator from Thales [4] has a limited memory size of 512 short integers per processor and the current $x86$ processors only holds 16 SSE registers. `Tesla` GPU cards have a global memory of 1 GB or more but the PCI-e $\times 16$ gen2 transfer rates are very slow compared to the CPU or GPU registers.

2.3 SIMD Processor Constraints

We are considering a particular class of hardware processors: the SIMD processors. They offer important speed-up due to their capability of executing the same instruction N times at once. They can be viewed as N Processing Element (PE) each executing the same code on different input. The implication on code control flow are well known:

1. Vector loops are critical;
2. Loop sizes matter;
3. Branches are to be avoided, though sometimes possible *via* masked execution.

The number of processor plays a key role, and is all the more important when masked execution is not possible.

2.4 Summary

In this section, we highlighted the three major concerns linked to the usage of hardware simd accelerators: the load-work-store model, the data transfers and the number of PE. We propose the algorithm in Alg. 1 to address them and generate automatically host and accelerator codes from a unique C code.

Algorithm 1 AcceleratorCodeGeneration(S, PE, M)

$S \leftarrow$ statement to optimize
$PE \leftarrow$ number of elementary processors
$M \leftarrow$ total memory size of the acceelrator
for all parallel loop nest l in S **do**
  if $depth(l) = 1$ **then**
    Strip mine l by PE
  end if{In case of a loop nest with $depth(l) \geq 2$, the outermost loop is selected}
  Declare a new variable N
  Apply loop expansion to the outermost loop of l so that the number of iterations is a multiple of PE {§ 4}
  Apply loop expansion to the innermost loop of l so that the number of iterations is a multiple of N {§ 4}
  Apply symbolic tiling to loop nest with tiling matrix $\begin{pmatrix} |PE| & 0 \\ 0 & N \end{pmatrix}$ {see [12]}
  Compute the memory footprint of the outermost tile loop of l as a polynomial $P(N, \sigma)$ {§ 5.2}
  Find the largest integer $e(\sigma)$ satisfying $P(N, \sigma) \leq N$ and add "$N = e$; before the external loop nest {§ 5.2}
  {Note: The code is still symbolic in N. N does not seem to go away in general. A new symbol N is needed for each loop nest in S}
  {Note: we have two host loops over the tiles and two tile loops to execute each tile. The outermost tile loop is the loop over the elementary processors.}
  Apply statement isolation on the outermost tile loop and generate memory transfers {§ 5.3}
  Apply outlining to the outermost tile loop to generate the host call code to the accelerator and to the innermost tile loop to generate the accelerator kernel {§ 6}
  Further process the kernel code to generate target's assembly {not detailed in this paper}
end for

The remaining of the paper is organized as follows: Section § 3 introduces array region concept, the next two (§ 4,§ 5) present innovative transformations which, combined with the traditional loop transformations loop tiling and parallel loop detection make it possible to generate efficient code for such heterogeneous machines. Separation between host and target is explained in § 6, § 7 applies the algorithm to a challenging example and § 8 concludes.

3 Interprocedural Array Regions

3.1 Definitions

Interprocedural convex array region analysis[8] is a powerful tool for program analysis. We will only give a quick overview of the main concepts here.

The convex array region for a variable v is computed for a program state σ at a statement S. It is denoted $\mathcal{R}^{S,\sigma}(v)$ and represents the set of indices of v accessed by S in state σ. That is $\mathcal{R}^{S,\sigma}(v) = \{v[\boldsymbol{\phi}] | C(\sigma).\boldsymbol{\phi} \leq 0\}$ where C is a constraints matrix depending on σ. We distinguish four kinds of regions, based on their access type:

Read Regions gather all elements used by S and are denoted $\mathcal{R}_r^{S,\sigma}(v)$
Write Regions gather all indices defined by S and are denoted $\mathcal{R}_w^{S,\sigma}(v)$
In Regions gather all elements whose initial values are used by S and are denoted $\mathcal{R}_{in}^{S,\sigma}(v)$
Out Regions gather all elements defined by S and used by one of its continuation and are denoted $\mathcal{R}_{out}^{S,\sigma}(v)$

By definition, we have $\mathcal{R}_{out}^{S,\sigma}(v) \subset \mathcal{R}_w^{S,\sigma}(v)$ and $\mathcal{R}_{in}^{S,\sigma}(v) \subset \mathcal{R}_r^{S,\sigma}(v)$. Each region can be labelled as may or must:

May Regions each region elements may be accessed or not (denoted, e.g. $\mathcal{R}_{r_{may}}^{S,\sigma}(v)$)
Must Regions each region element is always accessed (denoted, e.g. $\mathcal{R}_{r_{must}}^{S,\sigma}(v)$).

Convex array regions are defined by polyhedra, and under the assumption of a correct program, they are bounded by the hypercube defined by the variable definition space and thus define polytopes when the store is known.

In the following we use the $\overline{\ }$ operator to denote the smallest hypercube enclosing a polytope, and the $|.|$ operator to represent the volume of a region, given by an Ehrhart polynomial[2]. Note however that the volume of an hypercube is given by a regular polynomial only depending on the program state σ. The $\sqcup$ operator will represent the convex union of polyhedra.

We also introduce a new kind of region, the declaration region of a variable, denoted $R_{def}^{\sigma}(v)$. This region contains all possible legal indices of v. It depends of the store σ because of `C` *dependent types*.

3.2 Link with Hardware Constraints

Array regions offer a convenient formalism to model the behavior of a statement S. For any array $v \in S$:

Memory Footprint : $\sum_{v \in S} |\mathcal{R}_r^{S,\sigma}(v) \cup \mathcal{R}_w^{S,\sigma}(v)|$

Copy out elements: $\mathcal{R}_{out,may+must}^{S,\sigma}(v)$

Copy in elements: $\mathcal{R}_{in,may+must}^{S,\sigma}(v) \cup \mathcal{R}_{out,may}^{S,\sigma}(v)$. The reason why we include $\mathcal{R}_{out,may}^{S,\sigma}(v)$ is that an element of $\mathcal{R}_{out,may}^{S,\sigma}(v)$ may not have been modified by S and thus we have to ensure its value is correct by copying it *in*.

If we model accelerator memory as one big array m, we can summarize the constraint over the accelerator memory capacity using 3.2 into

$$\sum_{v \in S} |\mathcal{R}_r^{S,\sigma}(v) \cup \mathcal{R}_w^{S,\sigma}(v)| < |\mathcal{R}_{def}^{\sigma}(m)| \tag{1}$$

In Section 5, we show how to turn these equations into pieces code, using elementary transformations which are part of our algorithm (see Alg. 1).

4 Loop Expansion

We explained in Section 2 how the number of PE or the amount of available memory constrain our transformation.

We propose to *expand* the loop iteration set so that it matches the particular requirement of our hardware. This is different from array padding, because we really *add* statements in the execution flow. We can take two approaches when expanding a loop:

1. Guard the loop body so that no extra instruction, except the guard test, is executed. This is a conservative approach
2. Insert "new statement" in the execution flow. The legality of the insertion must be checked.

4.1 Guarding Loop Body

An option to enforce a given loop trip count is to guard the loop body with the initial iteration set. However such a solution is not ideal:

1. Branching is not always supported on SIMD architecture;
2. when supported through masking, it may degrade performance;
3. The guard evaluations are overhead.

We make a step forward if we distribute the guard on each statement of the body and examine each guarded statement for potential guard removal. A simple strategy here is to remove guards on statements that only write private variables, as shown by following example. Combined with scalarization, this strategy limits the number of guard inserted in the loop body. To go further and remove all guards, we have to consider a wider problem, the legality of inserting a new statement in the control flow.

```
int i,j,sz=10,a[sz];
for(i=0;i<sz+1;i++) {
  if(i<sz)  j=a[i]*a[i];
  if(i<sz)  a[i]=j;
}
```

```
int i,j,sz=10,a[sz];
for(i=0;i<sz+1;i++) {
  j=a[i]*a[i];
  if(i<sz)  a[i]=j;
}
```

4.2 Statement Insertion

Let S be a statement to be inserted in a program. What is the legality of such an insertion ? We must take care of

1. Syntactic validity: inserted statement has a valid syntax ;
2. Data-flow validity: inserted statement does not change the code data-flow;
3. Control-flow validity: inserted statement does not raise any exceptions.

In our case, Prop. 1 is guaranteed because we expand the loop iteration space: inserted statement already exist and its syntax is valid. To ensure Prop. 2, we use a pragmatic approach: once the statement is inserted, we compute its out regions. If they are empty, it asserts the statement does not change the data-flow. This approach is similar to the one used for loop-fusion: merge the loop and verify the dependency graph has not been altered.

Prop. 3 requires more care. We will focus on memory error and ignore arithmetic errors. Inserted statement may access data that have not been allocated, trough unmanaged subscripting. This subsection proposes a region based method to guarantee such access do not occur:

Let $\mathcal{R}_r(S)$ be the read region of S and $\mathcal{R}_w(S)$ its write region, as described in section 3. For each array variable v accessed in S, we also define $\mathcal{R}_{def}(v)$ as its declaration region. For example, the declaration region of **int** a[10][n] is $\{a[\phi_0][\phi_1] | 0 \leq \phi_0 < 10, 0 \leq \phi_1 < n\}$. Using this definition, we define a new variable v' whose declaration region is given by $\mathcal{R}_{def}(v') = \mathcal{R}_{def}(v) \sqcup \mathcal{R}_w(S) \cup \mathcal{R}_r(S)$ where $\sqcup$ is the convex union. Substituting v' to v in the declaration scope of v suppress memory errors caused by the insertion of S (which is in the scope of v). However this changes the type of v and may disturb the execution flow: in `C` the `sizeof` construct may create a dependence on the type of a variable, and pointer arithmetic also evaluate differently depending on the pointer type. In our implementation, we detect such construct and abort the insertion process if any. Likewise we do not perform inter-procedural type change and rely on inlining when such analysis would be needed.

4.3 Loop Expansion Algorithm

Finally, the loop expansion process is implemented as explained in Alg. 2, the inputs of which are a loop L_i and a number of iteration nb_iter

Algorithm 2 LoopExpansion(L_i, nb_iter)

```
1: copy body of L_i into statement S_dup
2: insert statement S_dup after L_i
3: if statement insertion on S_dup succeeds then
4:     remove S_dup
5:     perform loop expansion without guard
6: else
7:     remove S_dup
8:     perform loop expansion with guards
9: end if
```

5 From Array Region to User Code

5.1 Preamble

Before going into the details, we can get rid of the constraints related to the number of PE: In the tiling operation mentioned in Alg. 1, first tiling parameter must be PE.

5.2 Matching Size Constraints

We presented in Eq.1 the inequality that ties array regions and accelerator memory. Because of the simplification made in previous section, we can reword the inequality into:

$$\sum_{v \in S} |\overline{\mathcal{R}_r^{S,\sigma}(v) \sqcup \mathcal{R}_w^{S,\sigma}(v)}| < |\mathcal{R}_{def}^{\sigma}(m)| \tag{2}$$

However, there are no unknown in this inequality, which means it is either true or false. As a consequence we will introduce a new parameter t in σ, express Eq.2 as a function f of t and focus on finding a value of t that optimizes f.

The key transformation here is parametric tiling[12]. Picking a parallel loop l, we will tile it by a factor of t and compute parametric region of the tile. Note that combined with the statement in §5.1, this gives us a tiling matrix of the form $\begin{pmatrix} t & 0 \\ 0 & |PE| \end{pmatrix}$.

Yet tiling can be impossible, for example when a single parallel loop is considered. In that case we use two levels of strip mining as in following example:

```
for ( i =0; i<n ; i++)
   f ( i , S ) ;
```

which is transformed into:

```
for(k=0;k<n;k+=t*PE)
  for(j=0;j<PE;j++)
    for(i=k+j*t;i<MIN(k+(j+1)*t,n);i++)
      f(i,S);
```

That way we introduce a dependence over t in regions of S and compute them.

This results in the same inequality as Eq.2 with the unknown being $t \in \sigma$. For classical codes, f will be a linear expression of the form $a(\sigma) \times t + b(\sigma)$ from which we can easily deduce the exact integer value of t that optimizes Eq. 2, assuming $a(\sigma) \neq \infty$

$$t = \begin{cases} \lceil \frac{b(\sigma)}{a(\sigma)} \rceil \ | a(\sigma) < 0 \\ \lfloor \frac{b(\sigma)}{a(\sigma)} \rfloor \ | a(\sigma) > 0 \end{cases}$$

gives a state dependent formula for initializing t. If f is a higher level polynomial, we forward decision to a runtime function that uses polynomial root set and its highest degree coefficient sign to find an integer solution.

5.3 Statement Isolation

Dealing with accelerators means working in a separated memory and exchanging data between *host* and *accelerator* memories. The *statement isolation* transformation just does that: select a code statement and make sure all memory accesses in this statement are made to a new memory space allocated for this purpose. In order to keep consistency, data transfers are generated between this new memory and the old one. Below, we consider a program statement S in state σ and a array v referenced in S.

Memory Allocation From §3.2, we already know the memory footprint of v in S. The problem is to allocate in S a variable to hold the relevant part of $\mathcal{R}^{\sigma}_{def}(v)$. For the set gotten from the union in 3.2 may not be convex, we may need to allocate several variables. To avoid this difficulty (and at the cost of possible extra transfers) we use a convex union instead of a set union in the formula, ending up with a single replacement variable v' for which $\mathcal{R}^{\sigma}_{def}(v') = \mathcal{R}^{S,\sigma}_{r}(v) \sqcup \mathcal{R}^{S,\sigma}_{w}(v)$. And because `C` language enforces *declaration region* to be hypercubes, we end up with $\mathcal{R}^{\sigma}_{def}(v') = \overline{\mathcal{R}^{S,\sigma}_{r}(v) \sqcup \mathcal{R}^{S,\sigma}_{w}(v)}$. Note that such a definition does not guarantee that the smallest element in $\mathcal{R}^{\sigma}_{def}(v')$ is $(0, \ldots, 0)$, so we introduce a translation vector $\boldsymbol{t}$ to align the references to the v' declaration.

Data transfers Data transfers from v to v' and from v' back to v consist in direct translation of regions defined in 3.2 and 3.2. Such transfers can be generated using classical polyhedra scanning [1] combined with the translation vector $\boldsymbol{t}$. From the generated loop we can deduce the DMA call needed by the replacing innermost loops with equivalent `memcpy`.

Illustration Following code snippets show the result of region analysis on a simple tiled alphablending kernel: Only read and write array regions are represented as a set of constraints over array indices, as demonstrated in code 1.1

Listing 1.1: Convex array regions example

```
void alphablending(int n, short src0[n][n], short src1[n][n],
    short res[n][n]){
 unsigned int i, j;
 //PIPS generated variable
 unsigned int it, jt;
 for(it = 0; it <= n-1; it += 4)
  for(jt = 0; jt <= n-1; jt += 10)
// <result[PHI1][PHI2]-W-MAY-{it<=PHI1, PHI1<=it+4, PHI1+1<=n,
     jt<=PHI2, PHI2<=jt+10, PHI2+1<=n, 0<=it, 0<=jt, jt+1<=n}>
// <src0[PHI1][PHI2]-R-MAY-{it<=PHI1, PHI1<=it+4, PHI1+1<=n,
    jt<=PHI2, PHI2<=jt+10, PHI2+1<=n, 0<=it, 0<=jt, jt+1<=n}>
// <src1[PHI1][PHI2]-R-MAY-{it<=PHI1, PHI1<=it+4, PHI1+1<=n,
    jt<=PHI2, PHI2<=jt+10, PHI2+1<=n, 0<=it, 0<=jt, jt+1<=n}>
    for(i = it; i <= MIN(it+4, n-1); i += 1)
     for(j = jt; j <= MIN(jt+10, n-1); j += 1)
      result[i][j] = (40*src0[i][j]+(100-40)*src1[i][j])/100;
}
```

The result of *statement isolation* is shown in code 1.2: three new arrays have been allocated, three function calls take care of data movements and array indices are translated.

6 Separate Host and Accelerator Code

Once all size constraints have been met, we separate the host code, which performs the load-work-store calls, from the accelerator code, which performs the computation on isolated data.

This is done with the *outlining* transformation[17]: given a statement in an execution flow, it *outlines* the statement to a new procedure, adjusting call parameters to keep the execution flow correct. If we apply such a transformation to the loop over the PE we make the load-work-store triplet appear and have a new function to offload on the accelerator.

However the outlined function is not the function that will be executed by each PE: we have to outline the outlined function a second time to make it appear. Indeed the loop over the PE does not appear neither on host code nor on accelerator code, so we outline the body of the loop in a new function, the

Listing 1.2: Statement isolation on alphablending kernel

```
void alphablending(short src0[40][40], short src1[40][40],
    short result[40][40]) {
 unsigned int i, j;
 unsigned int i_t, j_t;
 for(i_t=0; i_t<=3; i_t+=1)
  for(j_t=0; j_t<=3; j_t+=1) {
   //PIPS generated variable
   short result0[10][10], src00[10][10], src10[10][10];
   copy_in(src10, 2,10,10, &src1[10*i_t][10*j_t],2,40,40);
   copy_in(src00, 2,10,10, &src0[10*i_t][10*j_t],2,40,40);
   for(i=10*i_t; i<=10*i_t+9; i+=1)
    for(j=10*j_t; j<=10*j_t+9; j+=1)
     result0[i-10*i_t][j-10*j_t]=(40*src00[i-10*i_t][j-10*j_t
         ]+60*src10[i-10*i_t][j-10*j_t])/100;
   copy_out(result, 2, 40, 40, &result0[10*i_t][10*j_t], 2,
       10, 10);
 }
}
```

accelerator code. This intermediate function must not appear in the final code, however its presence is critical in our process:

1. It preserves the sequential meaning of the code;
2. All transformation are given valid code as input;
3. It provides a *functional simulator* of the generated code.

Points 3 is especially important: even if we target a specific hardware accelerator, the set of compiled source keeps it functional behavior. As a consequence, a code transformed with the described series of transformations can be compiled and executed on regular architecture and debugged with classical tools.

7 Illustration

In this section, we use a running example 1.3 to illustrate all the steps of our algorithm, combining *statement isolation*, *symbolic tiling* and *hardware constraint equations* to meet the constraints of the accelerator.

Unlike [16], we tackle non static-control codes involving while loops, `gotos` and `structures`.

For this example, we will consider the Terapix processor [4] which has 128 PE and will assume an extended memory of 2^14 integers instead of the initial 2^8. We put no constraints on the number of registers.

Listing 1.3: Average power kernel

```
typedef struct { float re; float im;} Cplfloat;
float CplAbs(Cplfloat const * c) {
 return sqrtf(c->re*c->re+c->im*c->im);
}
void average_power(int Nth, int Nrg, int Nv, Cplfloat ptrin[
    Nth][Nrg][Nv], float Pow[Nth]) {
 int th, v, rg;
 for(th=0;th<Nth;++th)
  for (rg=0; rg<Nrg; rg++)
   for (v=0; v<Nv; v++)
    Pow[th]+=CplAbs(&ptrin[th][rg][v]);
}
```

7.1 Matching Size Constraints

It is possible to perform symbolic tiling [12]. From the number of PE in Terapix, we know the first loop is tiled by a factor of 128. The second tiling parameter is given by the solution to hardware constraints. Let $N \in \mathbb{N}$ be the unknown and the tiling matrix $\begin{pmatrix} 128 & 0 \\ 0 & N \end{pmatrix}$. We must enforce the outer loop trip count to be a multiple of the number of PE and perform *loop expansion*, as in code 1.4 the new loop upper bound is given by $\lceil \frac{x}{N} \rceil \times N$ that is, without the ceil operator, $\frac{x+N-1}{N} \times N$, where x is the loop trip count and N is the tiling parameter.

Listing 1.4: Average power expanded

```
for(th = 0; th <= 127; th += 1)
 for(rg = 0; rg <= N*((N+12)/N)-1; rg += 1)
  for(v = 0; v <= Nv-1; v += 1)
   Pow[th] += CplAbs(&ptrin[th][rg][v]);
```

Note that in this example, interprocedural constant propagation have substituted constant values 13 to symbolic values for `Nrg`.

We can now apply symbolic tiling with the insurance that all PE are always active:

Thanks to *loop expansion*, we have a perfect tiling, while we could have had an imperfect one. Note how we rely on `c99` feature to change array bound. The computation of regions for the loop statement iterating over the PEs gives us code 1.6

Listing 1.5: Average power tiled

```
for(tht = 0; tht <= 127; tht += 128)
 for(rgt = 0; rgt <= N*((N+12)/N)-1; rgt += N)
  for(th = tht; th <= MIN(tht+128, 127); th += 1)
   for(rg = rgt; rg <= MIN(rgt+N, N*((N+12)/N)-1); rg += 1)
    for(v = 0; v <= Nv-1; v += 1)
     Pow[th] += CplAbs(&ptrin[th][rg][v]);
```

Listing 1.6: Regions analysis after tiling

```
// <Pow[PHI1]-R-MAY-{PHI1<=127, tht<=PHI1, 0<=tht, tht<=127}>
// <Pow[PHI1]-W-MAY-{PHI1<=127, tht<=PHI1, 0<=tht, tht<=127}>
// <ptrin[PHI1][PHI2][PHI3][.im]-R-MAY-{PHI1<=127, tht<=PHI1,
    PHI2<=N+rgt, rgt<=PHI2, 0<=PHI3, PHI3+1<=Nv ...}>
// <ptrin[PHI1][PHI2][PHI3][.re]-R-MAY-{PHI1<=127, tht<=PHI1,
    PHI2<=N+rgt, rgt<=PHI2, 0<=PHI3, PHI3+1<=Nv ...}>
for(th = tht; th <= MIN(tht+128, 127); th += 1)
 for(rg = rgt; rg <= MIN(rgt+N, N*((N+12)/N)-1); rg += 1)
  for(v = 0; v <= Nv-1; v += 1)
   Pow[th] += CplAbs(&ptrin[th][rg][v]);
```

From which we compute memory usage and find the volume polynomial:

$$V(tht, Nv) = 4\times(128-tht)+4\times(((128-tht)\times Nv)\times N+((128-tht)\times Nv))-2^{14}$$

Which has a rational solution $N = \frac{128\times Nv-((Nv+1)\times tht+3968)}{Nv\times tht-128\times Nv}$ Because the polynomial is of degree 1 and tends toward $+\infty$, taking the ceil of previous formula is a satisfying solution and we can now substitute it to the symbolic value of N and perform transfer generation.

7.2 Memory Transfer Generation

Size constraints are met and we generate data transfers between the loop over the PEs and the remaining of the code. We use *statement isolation* to do so. From the computed regions, we generate temporary arrays that will represent the isolated memory and load - store loops that will transfer data. So after *statement isolation*, we get the code in code 1.7: Last step is to perform the separation between host code and accelerator code.

7.3 Separate Host and Accelerator Code

Applying outlining as exposed in Section 6 results in code 1.8: This example demonstrates that, unlike traditional outliners, our version takes advantage of several features from `C` language: dependent types from `C99` and pointer to array sections are used.

Listing 1.7: Average power after statement isolation

```
for(rgt = 0; rgt <= N*((N+12)/N)-1; rgt += N) {
 //PIPS generated variable
 float Pow0[-tht+127+1];
 //PIPS generated variable
 Cplfloat ptrin0[-tht+127+1][N+1][Nv];
 /* transfer loop generated by PIPS from ptrin to ptrin0 */
 for(i2 = 1; i2 <= Nv; i2 += 1)
  for(i3 = 1; i3 <= 1+N; i3 += 1)
   for(i4 = 1; i4 <= -(-128+128*tht)+128; i4 += 1)
    ptrin0[-1+i4][-1+i3][-1+i2] = ptrin[-128+128*tht+-1+i4][
        rgt+-1+i3][-1+i2];
 /* transfer loop generated by PIPS from Pow to Pow0 */
 for(i0 = 1; i0 <= -(-128+128*tht)+128; i0 += 1)
  Pow0[-1+i0] = Pow[-128+128*tht+-1+i0];
 /* real computation */
 for(th = 1; th <= MIN(-128+128*tht+128, 127)-(-128+128*tht)
     +1; th += 1)
  for(rg = 1; rg <= MIN(rgt+N, N*((N+12)/N)-1)-rgt+1; rg += 1)
   for(v = 1; v <= Nv; v += 1)
    Pow0[-1+th+-128+128*tht-(-128+128*tht)] += CplAbs(&ptrin0
        [-1+th+-128+128*tht-(-128+128*tht)][-1+rg+rgt-rgt][-1+
        v-0]);
 /* transfer loop generated by PIPS from Pow0 to Pow */
 for(i1 = 1; i1 <= -(-128+128*tht)+128; i1 += 1)
  Pow[-128+128*tht+-1+i1] = Pow0[-1+i1];
}
```

8 Conclusion

This paper presents a generic source-to-source code transformation algorithm that can be used to meet the hardware constraints generally found in simd accelerators [20].

- *statement isolation* generates data transfers from convex array regions,
- *loop expansion* offers an efficient alternative to *loop peeling*,
- *outlining* separates host and accelerator code,

Throughout this paper, inter-procedural *convex array regions* are used as an intuitive and powerful abstraction to analyze code behavior and memory footprint. The approach is demonstrated on an unusual running example, involving reductions, symbolic bounds structure fields and function call, showing the impact of an accurate analysis in the presence of complex constructs.

We are currently working on specializing our model for CUDA code generation, taking into account complex memory hierarchy. A limitation of the approach is that we use a synchronous model for data transfers. This fails to model

Listing 1.8: Code split in three functions

```
/* call to kernel*/
work_0(N, Nv, MIN(N+rgt, N*((N+12)/N)-1)-rgt+1, Pow0, ptrin0);
/*...*/
void work_0(int N, int Nv, int I_3, float Pow0[128], Cplfloat
    ptrin0[128][N+1][Nv]) {
 int rg, th, v;
 for(th = 1; th <= 128; th += 1)
  work_0_kernel(N, Nv, I_3, &Pow0[-1+th], &ptrin0[-1+th]);
}
void work_0_kernel(int N, int Nv, int I_3, float *Pow00,
    Cplfloat (*ptrin00)[N+1][Nv]) {
 int rg, v;
 for(rg = 1; rg <= I_3; rg += 1)
  for(v = 1; v <= Nv; v += 1)
   *Pow00 += CplAbs(&((*ptrin00)[rg-1])[v-1]);
}
```

important optimizations such as double buffering. Memory footprint and data transfers generations needs some extensions to take care of it.

Acknowledgements

Thanks a lot to members of the Thales-CRT for their continuous help and advices that made interfacing two worlds – software and hardware – less painful. Special thanks to Béatrice Creusillet, whose work on extension of convex array regions to the `C` language proved to be invaluable. This work is funded by the ANR FREIA project number ANR-07-ARFU-004-02.

References

1. Ancourt, C., Irigoin, F.: Scanning polyhedra with do loops. In: PPOPP. pp. 39–50 (1991)
2. Barvinok, A.I.: A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. In: FOCS. pp. 566–572 (1993)
3. Bastoul, C.: Improving Data Locality in Static Control Programs. Ph.D. thesis, University Paris 6, Pierre et Marie Curie, France (Dec 2004)
4. Bonnot, P., Lemonnier, F., Edelin, G., Gaillat, G., Ruch, O., Gauget, P.: Definition and SIMD implementation of a multi-processing architecture approach on FPGA. In: Design Automation and Test in Europe 2008. pp. 610–615. IEEE CEDA, ACM SIGDA, IEEE (December 2008)
5. CAPS: Hmpp workbench, `http://www.caps-entreprise.com/hmpp.html`
6. Clauss, P.: Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs (1996)

7. Coleman, S., Kinley, K.S.M.: Tile size selection using cache organization and data layout. pp. 279–290. ACM Press (1995)
8. Creusillet, B., Irigoin, F.: Interprocedural array region analyses. Int. J. Parallel Program. 24(6), 513–546 (1996)
9. Genest, G., Chamberlain, R., Bruce, R.J.: Programming an fpga-based super computer using a c-to-vhdl compiler: Dime-c. In: AHS. pp. 280–286 (2007)
10. Guo, Z., Najjar, W., Buyukkurt, B.: Efficient hardware code generation for fpgas. ACM Trans. Archit. Code Optim. 5(1), 1–26 (2008)
11. Han, T.D., Abdelrahman, T.S.: hicuda: a high-level directive-based language for gpu programming. In: GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units. pp. 52–61. ACM, New York, NY, USA (2009)
12. Hartono, A., Baskaran, M.M., Bastoul, C., Cohen, A., Krishnamoorthy, S., Norris, B., Ramanujam, J., Sadayappan, P.: Parametric multi-level tiling of imperfectly nested loops. In: ICS '09: Proceedings of the 23rd international conference on Supercomputing. pp. 147–157. ACM, New York, NY, USA (2009)
13. Irigoin, F., Jouvelot, P., Triolet, R.: Semantical interprocedural parallelization: An overview of the PIPS project,. In: 1991 International Conference on Supercomputing, Cologne, June 1991 (1991)
14. Lam, M.S., Rothberg, E.E., Wolf, M.E.: The cache performance and optimizations of blocked algorithms. In: In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 63–74 (1991)
15. Lee, S., Min, S.J., Eigenmann, R.: Openmp to gpgpu: a compiler framework for automatic translation and optimization. In: PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 101–110. ACM, New York, NY, USA (2009)
16. Leung, A., Vasilache, N., Meister, B., Baskaran, M., Wohlford, D., Bastoul, C., Lethin, R.: A mapping path for multi-gpgpu accelerated computers from a portable high level programming abstraction. In: GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. pp. 51–61. ACM, New York, NY, USA (2010)
17. Liao, C., Quinlan, D.J., Vuduc, R., Panas, T.: Effective source-to-source outlining to support whole program empirical optimization. In: Proc. Int'l. Wkshp. Languages and Compilers for Parallel Computing (LCPC). vol. LNCS. Newark, DE, USA (October 2009)
18. NVIDIA: NVIDIA CUDA Reference Manual 2.3. `http://www.nvidia.com/object/cuda_develop.html` (July 2009)
19. Ueng, S.Z., Lathara, M., Baghsorkhi, S.S., Hwu, W.M.W.: CUDA-Lite: Reducing GPU Programming Complexity, pp. 1–15. Springer-Verlag, Berlin, Heidelberg (2008)
20. Wolfe, M.: Implementing the PGI accelerator model. In: GPGPU. pp. 43–50 (2010)