Compilation for Heterogeneous Computing: Automating Analyses, Transformations and Decisions

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

[1] Institut Télécom, Télécom Bretagne, Info/HPCAS, Plouzané, France
[2] MINES ParisTech, CRI, Fontainebleau, France
[3] HPC Project, Meudon, France

Abstract. Hardware accelerators, such as FPGA boards or GPU, are an interesting alternative or a valuable complement to classic multi-core processors for computational-intensive software. However it proves to be both costly and difficult to use legacy applications with these new heterogeneous targets. In particular, existing compilers are generally targeted toward code generation for sequential processors and lack the required abstractions and transformations for automatic code generation and code re-targeting for heterogeneous targets. The goal of this article is to introduce a set of high-level code transformations based on an abstraction of existing hardware architectures that make it possible to build compilers specific to a target using a shared infrastructure. These transformations have been used to build two completely automatic compilers for an FPGA-based embedded processor and an NVIDIA GPU. The latter is validated on several representative digital signal processing kernels.

1 Introduction and Context

The two main approaches to accelerate software currently both involve parallelism: classic multi-cores processors and hardware accelerators. The latter trades a limited class of applications for a potentially more interesting speedup per Euro and speedup per watt ratios. However, such hardware accelerators suffer from typical limitations:

- developing cost of new applications;
- difficulty and cost of porting legacy code (entry cost);
- multiple sources as a consequence of target diversity;
- cost of switching from on architecture to another, i.e. cost of performance or programming portability (exit cost).

In order to make a durable investment in an accelerator, it is essential to free the user from these constraints: an automated approach is a natural way to leave him with a single source code to maintain. The compiler is in charge of generating architecture-specific code.

Besides, automatic and semi-automatic code generation techniques for hardware accelerators have received a growing attention over the last years, with the obvious goal of filling the gap between the increased performance offered by graphics cards (GPU) and their lack of programmability. The fact that 4 heterogeneous machines find themselves in the top

10 of the Top500 of November 2010, with the Tianhe-1a cluster using Fermi GPUs from NVIDIA ranking first, is a good illustration of this focus. NVIDIA proposes the CUDA [16] language, an extension of the C89 language with a C++ flavour, and a compiler, `nvcc`, to generate code targeted to their GPU. It certainly limits the number of targeted accelerators, and requires a recoding of computational-intensive code sections. *A contrario* OpenCL [12] offers an API and a language extending C99, comparable to CUDA, that provides a hardware abstraction to the expense of a lower-level API. It ensures code portability but does not guarantee performance portability yet. As an example, [11] shows on a set of kernels that an OpenCL implementation can run from 10 to 40% slower than the equivalent CUDA code, including transfer time.
CAPS Entreprise [4] has developed the HMPP compiler which uses a directive-based approach: all parameters required for code generation must be provided, but additional optimizations are possible. In [14] authors have similarly used the information from OpenMP directives as a starting point for CUDA code generation. PGI's compiler [19] takes a similar semi-automatic approach where the user flags computational kernels by using also directives inserted in original code, but in a more automatic way. Optionally, additional directives can be added to take over from compiler analyses when they are not accurate enough. In that case, the user benefits from an incremental approach.
In spite of several similarities on the host side, code generation for FPGAs has experienced diverging developments: the `c2h` compiler, from Altera, or Mitrion-C [13] both take a subset of the C language to generate VHDL. [6,7,1] give many insights into the potential optimizations to use in order to improve the behavior of such tools.
In this context, it seems unrealistic to try to offer a single tool that takes a unique source code as input, even with annotations, and generates as many outputs as the number of targeted hardware. In this article, we propose an approach based on the analysis of the various constraints exerted on the input code by an architecture, e.g. the accelerator memory size or the number of processing elements (PE). Using the compiler infrastructure as a toolbox, the compiler developer builds a compilation chain so that all the constraints are satisfied by the refined code. A hardware compiler, supplied by the hardware vendor, plays the part of the back end. This binding between constraints and transformations enforces transformation reuse and provides a generic methodology to builds a compiler for new heterogeneous targets at low expense.
To illustrate our proposal, the analyses and code transformations presented in this article are demonstrated on a running example, typical from signal processing applications; a horizontal erosion, typically used in image processing applications (see Figure 1). All transformations introduced in this paper are implemented in the PIPS [9,10] source-to-source compilation infrastructure. Exhibited code fragments are presented without any post-processing other than cosmetic.
This document is organized as follows. Section 2 introduces our target model. Section 3 proposes a statement computation intensity estimation method. Section 4 describes the outlining step applied to extract kernel code. Section 5 presents the data transfer generation algorithm and Sec-

```
void erode(int n, int m, int in[n][m], int out[n][m]) {
  for(int i=0;i<n;i++)
    for(int j=0;j<m;j++)
      if( j==0 ) out[i][j]=MIN(in[i][j],in[i][j+1]);
      else if(j==m-1) out[i][j]=MIN(in[i][j-1],in[i][j]);
      else out[i][j]= MIN(in[i][j-1],in[i][j],in[i][j+1]);
}
```

Fig. 1: Horizontal erosion code sample.

tion6 proposes a technique to take memory size constraints into account. Experimental results are presented in Section 7.

2 Target Model

Both graphics cards or FPGA boards are part of hybrid computers and use a master-worker paradigm: the *host* processor offloads computational-intensive functions to a remote *accelerator* using a call sequence similar to load - work - store. This sequence is typical of hybrid computations and the proposals of this paper are made on its basis.
Data transfers, *load*s and *store*s are performed using copy operators, or DMAs, which can be either synchronous or asynchronous, depending on the target hardware. They can transfer n-dimensional blocks of data, where n is also architecture dependant (e.g. $n = 1$ in CUDA). Additionally, some specific constraints can exist, such as data alignment or maximal transfer size. The memory size also impacts this sequence because it is not possible to transfer more cumulative data than the amount of memory available on the accelerator. The speedup can come from a specific implementation of some libraries, e.g. a ClearSpeed Advance board accelerating the Intel Math Kernel library, or from a massive usage of parallelism found in GPUs or FPGA. Table 2 summarizes some significant constraints relevant to hybrid computing for two targets: the embedded vector processor Ter@pix [3] from THALES and a NVIDIA GPU Quadro FX 2700M chosen in the lower end of the market to have a comparable electrical consumption. Some additional constraints must be taken into

	Quadro FX 2700M	Ter@pix
Parallelism type	±SIMD	SIMD
Bus bandwidth B (byte/s)	5G (PCI)	1.2G (DDR 2)
Number of PE	48	128
DMA dimension	1d	2d
DMA size	no constraint	$k \times 128$
Accelerator memory (*bytes*)	512 M (global RAM)	1024
Instruction Set Architecture	PTX	Terasm

Fig. 2: Characteristics of Quadro and Ter@pix accelerators.

account. However they are too target-dependent and thus not valuable candidate for hardware abstraction.

3 Computational Intensity Estimation

The first step to automatically generate accelerator code is to identify the code fragments candidate for offloading. Depending on the memory - operation ratio and the accelerator kind, offloading can lead to a speedup or a slowdown. Let t_n be the execution time of a kernel carried by statement S in a memory state σ modelled by Equation (1),

$$t_n(S,\sigma) = \tau_0 + \frac{V(S,\sigma)}{B} + \frac{t_{\mathrm{s}}(S,\sigma)}{a_{\mathrm{th}}} \tag{1}$$

where $V(S,\sigma)$ is the memory footprint of statement S depending on σ and B is the average memory bandwidth between the host and the accelerator introduced in Table 2. τ_0 is a warming up cost, e.g. a DMA initialization delay; $t_{\mathrm{s}}(S,\sigma)$ is the sequential execution time of S in state σ on the host side, depending on σ; and a_{th} is the expected average acceleration provided by the accelerator, dependent of the number of PE, the parallelism type and the ISA of both the host and the accelerator. For the offloading to be relevant, the inequality $t_n(S,\sigma) < t_{\mathrm{s}}(S,\sigma)$ must be satisfied, that is condition (2), which turns into (3), must be met. In that case, the code can be profitably offloaded.

$$\tau_0 + \frac{V(S,\sigma)}{B} < t_{\mathrm{s}}(S,\sigma) \times \frac{a_{\mathrm{th}} - 1}{a_{\mathrm{th}}} \tag{2}$$

$$(\frac{1}{B} + \frac{\tau_0}{V(S,\sigma)}) \times \frac{a_{\mathrm{th}}}{a_{\mathrm{th}} - 1} < \frac{t_{\mathrm{s}}(S,\sigma)}{V(S,\sigma)} \tag{3}$$

An estimation of $t_{\mathrm{s}}(S,\sigma)$ can be computed statically in many cases using the execution model described in [20]. When applicable, this method recursively associates a polynomial function of σ to each statement S in state σ. An approximation of $V(S,\sigma)$ can be obtained from convex array region analysis [5]. It gives exact or over-approximated information about the set of data read or written by S represented as a system of linear inequalities. It is possible to compute the volume $V(S,\sigma)$ of the associated polyhedron as a polynomial [2].
Figure 3 shows the result of the implementation of these analyses in PIPS for the code from Figure 1. $t_{\mathrm{s}}(S,\sigma) = 20.25 \times m \times n + 4 \times n + 3$ and $V(S,\sigma) = 2 \times m \times n$ are found for the sequential time and memory footprint respectively, the ratio of which tends to 10.125 when $m \to \infty$ and $n \to \infty$. The ratio $\frac{t_{\mathrm{s}}(S,\sigma)}{V(S,\sigma)}$ may not be statically known. In that case, a test can be inserted to dynamically select the sequential or accelerated code version, based on the runtime evaluation of its expression.
Table 4 demonstrates the usefulness of this analysis for automatic GPU code generation. Using the same experimental parameters as in Section 7.2, "`gcc -O3`" and "`p4a --cuda`" have been used to generate a CPU and a GPU binary, respectively. Input code is a finite impulse response filter kernel with hard coded window size, and thus the computational

```
//    20.25*m.n + 4*n + 3 (SUMMARY)
void erode(int n, int m, int in[n][m], int out[n][m])
```

(a) Complexity Analysis

```
//   <in[PHI1][PHI2]-R-MAY-{0<=PHI1, PHI1+1<=n, 0<=PHI2, 1<=
    PHI2+m,
//     PHI2<=m, 1<=m}>
//   <out[PHI1][PHI2]-W-MAY-{0<=PHI1, PHI1+1<=n, 0<=PHI2, PHI2
    +1<=m}>
void erode(int n, int m, int in[n][m], int out[n][m])
```

(b) Array Region Analysis

Fig. 3: Complexity and array region analyses on a horizontal erosion code.

Window size	1	2	4	8	16	32	64	128	256	512	1024
CPU time (s)	1.55	1.59	1.67	1.82	2.13	2.73	4.00	6.70	11.50	21.57	41.31
GPU time (s)	2.51	2.51	2.53	2.56	2.61	2.73	2.98	3.38	4.01	4.99	5.69

Fig. 4: CPU and GPU execution time for various FIR window sizes.

intensity increases with the window size. In that case, the $\frac{t_s(S,\sigma)}{V(S,\sigma)}$ ratio is a constant. Until the window size reaches the value of 32, the transfer time dominates GPU execution times. Afterwards, the computational intensity is large enough to achieve some speedup. As a consequence, choosing a good $\frac{a_{\mathrm{th}}}{B(a_{\mathrm{th}}-1)}$ makes it possible to select the loops that are relevant for offloading and those that are not, which is only possible if profiling information provide reasonable experimental values for a_{th} and B.

4 Function Outlining

Once computationally intensive code fragments have been identified, it is necessary to turn the associated statements into function calls to separate the host code from the accelerator code.
If the outlining process guarantees that outlined functions use neither global variables nor return values, it is also a mean to isolate variables local to the accelerators from those declared in the outlined functions and those used to share data between the caller and the callee, i.e. between the host and the accelerator.
The outlining process is based on Formula (4) adapted from [15]

$$
\begin{aligned}
ExternalVars(S) &= ReferencedVars(S) \quad (4)\\
&\quad -(DeclaredVars(S) \cup PrivateVars(S)) \quad (5)
\end{aligned}
$$

where S is a statement; $ExternalVars$ is the set of variables passed as parameters; $ReferencedVars(S)$ gathers all variables referenced by

S; $DeclaredVars(S)$ gathers all variables locally declared in S and $PrivateVars(S)$ is the set of global variables that have been privatized to S. $ReferencedVars$ and $DeclaredVars$ are built through a recursion over the IR of S and $PrivateVars$ come as a result of a privatizing phase such as [18].
During the outlining process, it is important to take into account type dependencies, in particular those implied by variable-size arrays from C99. Thus, the set $ExternalVars(S)$ must be increased by the set of all variables required by the type definition of each variable it contains. A function outlined in this manner is a suitable candidate for an OpenCL kernel, because it is a self-contained function.
Figure 5 illustrates this process on the example from Figure 1, in which the internal loop is outlined as a new function `kernel`.

```
void erode(int n, int m, int in[n][m], int out[n][m]) {
  for(int i = 0; i <= n-1; i += 1)
    kernel(m, n, i, in, out);
}
void kernel(int m, int n, int i, int in[n][m], int out[n][m])
{
  for(int j = 0; j <= m-1; j += 1)
    if (j==0) out[i][j] = MIN(in[i][j], in[i][j+1]);
    else if (j==m-1) out[i][j] = MIN(in[i][j-1], in[i][j]);
    else out[i][j] = MIN(in[i][j-1], in[i][j], in[i][j+1]);
}
```

Fig. 5: Outlining of an erosion kernel.

5 Data Transfer Generation

Once a kernel is found and outlined to a separate function, it is necessary to generate data transfers and memory allocations to perform the remote procedure call. To do so, a transformation called **statement isolation** is introduced.
Given a statement S, it is possible to compute an estimate of the array regions read or written by this statement for each array variable v referenced by S. These regions are denoted $\mathcal{R}_r(v)$ and $\mathcal{R}_w(v)$, respectively. Depending on the accuracy of the analysis, these regions are either exact, $\mathcal{R}_r^{=}(v)$, or over-estimated, $\mathcal{R}_r^{\succeq}(v)$. There is a strong binding between these array regions and data transfers:

Transfers from the accelerator All data that may be written by S must be copied back to the host from the accelerator:

$$\mathcal{T}_{H\leftarrow A}(S) = \{\mathcal{R}_w(v) \mid v \in S\} \qquad (6)$$

```
void erode(int n, int m, int in[n][m], int out[n][m]) {
  int (*out0)[n][m] =  0, (*in0)[n][m+1] =  0;
  P4A_accel_malloc((void **) &in0, sizeof(int)*n*(m+1));
  P4A_accel_malloc((void **) &out0, sizeof(int)*n*m);
  P4A_copy_to_accel_2d(sizeof(int), n, m, n, m+1, 0, 0, &in
      [0][0], *in0);
  P4A_copy_to_accel_2d(sizeof(int), n, m, n, m, 0, 0, &out
      [0][0], *out0);
  for(int i = 0; i <= n-1; i += 1)
    for(int j = 0; j <= m-1; j += 1)
      if (j==0) (*out0)[i][j] = MIN((*in0)[i][j], (*in0)[i][j
          +1]);
      else if (j==m-1) (*out0)[i][j] = MIN((*in0)[i][j-1], (*
          in0)[i][j]);
      else (*out0)[i][j] = MIN((*in0)[i][j-1],(*in0)[i][j],(*
          in0)[i][j+1]);
  P4A_copy_from_accel_2d(sizeof(int), n, m, n, m, 0, 0, &out
      [0][0], *out0);
  P4A_accel_free(in0);
  P4A_accel_free(out0);
}
```

Fig. 6: Code after statement isolation.

Transfers to the accelerator All data that may be read by S must be copied from the host to the accelerator:

$$\mathcal{T}_{H\to A}(S) = \{\mathcal{R}_r(v) \,|\, v \in S\}$$

Indeed, all data for which we have no guarantee of write by S must be copied in. Otherwise, uninitialized data may be transfered back to the host without being initialized. So the extended formula is:

$$\mathcal{T}_{H\to A}(S) = \{\mathcal{R}_r(v) \,|\, v \in S\} \cup \{\mathcal{R}_w^{\succeq}(v) \,|\, v \in S \;\wedge\; \nexists \mathcal{R}_w^{=}(v)\} \qquad (7)$$

Based on Equations (6) and (7) it is possible to allocate new variables on the accelerator, to generate copy operations from the old variables to the newly allocated ones and to perform the required change of frame. Figure 6 illustrates this transformation on the running example.
It presents the variable replacement, the data allocation and the 2D data transfers; the latter are delegated to a runtime, in that case Par4All's [8]. Its interface is target-independent and the implementation is specialized depending on the targeted accelerator.

6 Memory Constraints

Many embedded accelerators have a limited memory size. If this size is not sufficient to run the whole computation in a single pass, the latter must be split in chunks: e.g. the Ter@pix accelerator is dedicated to

image processing but its memory cannot hold a full image (see Table 2), so all the processing is done on tiles.
In order to fulfill this memory constraint, the iteration space of the computational kernels must be tiled. Let us assume the kernel is in the form of a perfectly nested loop S of depth n. In order to work out the tiling parameters, a two-step process is used: n symbolic values denoted $p_1, \ldots, p_n$ are introduced to represent the computational blocks and a symbolic tiling, parameterized by these values, is performed. It generates n outer loops and n inner loops. The statement carrying the inner loops is denoted S_{inner} and the memory state before its execution is denoted σ_{inner}. The idea is to run the inner loops on the accelerator once the p_k are chosen so that the memory footprint of S_{inner} does not exceed a threshold fixed by the hardware. To this end, the memory footprint $V(S_{\text{inner}}, \sigma_{\text{inner}})$ is computed and a solution for $p_1, \ldots, p_n$ satisfying Condition (8) is searched.

$$V(S_{\text{inner}}, \sigma_{\text{inner}}) \leq V_{\max} \tag{8}$$

$V_{\max}$ is the memory size of the considered accelerator. This gives an inequality over the p_k. Other constraints can be gathered from the accelerator model specified in Section 2: E.g. a vector accelerator will require p_1 to be set to the vector size.
Figure 7 shows the effect of a symbolic tiling and the result array region analysis on the running example. As a result, the memory footprint of S_{inner} is given as a function of p_1, p_2 in Equation (9).

$$V(S_{\text{inner}}, \sigma_{\text{inner}}) = 2 \times p_1 \times p_2 \tag{9}$$

In the case of Ter@pix, we must set $p_1 = 128$, where 128 is the number of processing elements, and $V_{\max} = 1024$ for each node. It gives a direct expression for p_2: $p_2 = \frac{1024 \times 128}{128 \times 2} = 512$.

7 Applications

The transformations presented in the sections supra have been used to build two compilers for two drastically different accelerators, a FPGA-based embedded processor specialized in signal processing developed by THALES, Ter@pix [3]; and an NVIDIA GPU. The former compiler is a research project developed in the FREIA project funded by the French ANR. The latter is an open source product developed by the HPC Project company. Both compilers use the basic building blocks described in the above sections for the host side code generation. They rely on specific transformations to take other constraints into account and on the vendor compiler for the actual binary code generation.

7.1 Ter@pix

This accelerator has been designed to process images from a camera at run time. It can run signal processing and morphological mathematics

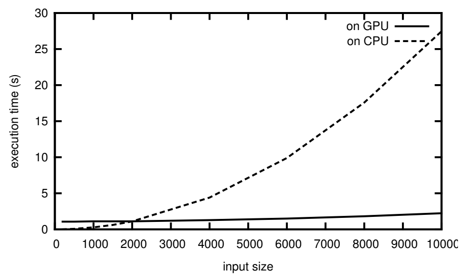
```
void erode(int n, int m, int in[n][m], int out[n][m]) {
  int p_1, p_2;
  for(int it = 0; it <= n-1-(p_1-1); it += p_1)
    for(int jt = 0; jt <= m-1-(p_2-1); jt += p_2)
// <in[PHI1][PHI2]-R-MAY-{it<=PHI1, PHI1+1<=it+p_1, PHI1+1<=n
//  jt<=PHI2+1, PHI2<=jt+p_2, 2jt+1<=PHI2+m, PHI2<=m, 1<=p_2,
     0<=m, 0<=n}>
// <out[PHI1][PHI2]-W-MAY-{it<=PHI1, PHI1+1<=it+p_1, PHI1+1<=
    n
//   jt<=PHI2, PHI2+1<=jt+p_2, PHI2+1<=m, 0<=m, 0<=n}>
      for(int i = it; i <= MIN(it+p_1, n-1+1)-1; i += 1)
        for(int j = jt; j <= MIN(jt+p_2, m-1+1)-1; j += 1)
          if (j==0) out[i][j] = MIN(in[i][j], in[i][j+1]);
          else if (j==m-1) out[i][j] = MIN(in[i][j-1], in[i][
              j]);
          else out[i][j] = MIN(in[i][j-1], in[i][j], in[i][j
              +1]);
}
```

Fig. 7: Symbolic tiling of the outermost loop of an horizontal erosion.

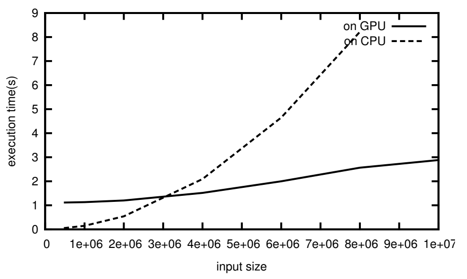
kernels, called **microcode**s, combined in various ways. Current development work-flow involves the manual writing of the microcodes in a specific assembly language for the accelerator side, and the chaining of the host-side calls in C. The basic operations are classified as follows: point-to-point operators – sum of images, maximum of images, etc; stencil operators such as erosion, dilatation and convolution; reducing operators like average power; and those involving an indirection, for instance an histogram.
A compiler prototype called `terapyps` has been developed for this target, using the chaining of outlining, symbolic tiling and statement isolation described in this article. It handles the first two classes of microcodes. Reducing operators are currently implemented using typical reduction parallelization to come down to the first cases. Microcodes involving an indirection cannot be handled by the array region analysis and, for this reason, they are not covered by this approach.
There is no hardware compiler for this target, therefore the assembly code generation has been implemented as part of the original compiler. Both host-side and accelerator-side code generation are handled.
However, most experimentation efforts have focused on the GPU compiler presented in the next section, so experimental results in terms of speedup compared to manual implementation are not available yet.

7.2 GPU

Although approaches such as CUDA and OpenCL make it easier to develop on GPU, they are far from being automatic. The Open Source Par4All [8] compiler, a C-to-CUDA compiler, has been developed by HPC

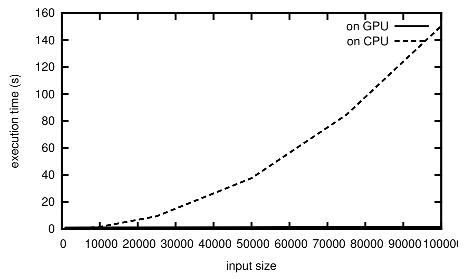


(a) Convolution

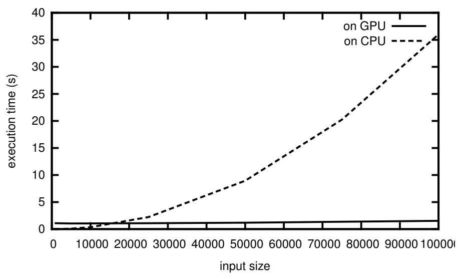


(b) FIR

Fig. 8: Median execution time on a GPU for image processing kernels.



(a) N-Discrete Time Fourier Transform



(b) Correlation

Fig. 9: Median execution time on a GPU for DSP kernels.

Project using the transformations described in this article for the host side, and GPU-specific transformations for the accelerator side. Those transformations go beyond the scope of this article and will not be further detailed. The interested reader can however consult the publicly available 1K lines of code of the `p4a` processor for a deeper insight. Symbolic tiling is not integrated yet into the tool, because it is less needed for GPU cards for which the global memory is far larger than Ter@pix's. The tool validation suite includes the following image processing kernels: a convolution, with a window size of 5, and a finite impulse response filter, with a window size of $\frac{n}{1000}$. The erosion code did not pass the computation intensity test on the considered machine. Execution times are given in Figure 8.

Measurements have been made using a desktop station hosting a 64-bit Debian/testing with GCC 4.3.5 and a 2-core 2.4 GHz Intel Core2 CPUs. The CUDA 3.2 compiler is used and the generated code is executed on a Quadro FX 2700M card. Sources are compiled using the command `p4a --cuda` ***`input_file.c`*** `-o` ***`binary`***. Compilation is fully automatic. The development version of PAR4ALL is used, linked with the development version of PIPS. The whole run is measured, i.e. timings

include GPU initialization, data transfers, kernel calls, etc. The median over 100 runs is taken. Figure 9 shows additional results for digital signal processing kernels extracted from [17] and available on the website `http://www.ece.rutgers.edu/~orfanidi/intro2sp`: a N-Discrete Time Fourier Transform and a sample cross correlation.
These results are promising and show that an automatic approach to GPU code generation is possible. Results get even better with a newer card with better computation and memory performances. The main advantage of our approach is the source-level compatibility, without directive support.

8 Conclusion

The contributions of this article are the definitions of four transformations corresponding to that many hardware constraints on the original source code, candidate to hybrid computing. **Computational intensity estimation** is used to take into account the trade-off between execution time and transfer time; **outlining** isolates a statement in a new function; **statement isolation** sets up a separate memory space for the kernel; and **symbolic tiling** coupled to **memory footprint computation** handles accelerator memory size limitation.
These transformations have been used to build two automatic compilers for the C language, `terapyps` and `p4a`, for an FPGA-based embedded processor and NVIDIA GPU, respectively. The `p4a` [8] compiler has been validated on various signal processing kernels and other scientific applications, demonstrating that an automatic approach is feasible.
On-going work includes interprocedural data transfer optimizations and generation of asynchronous transfers to mask transfer times. In particular, the impact of this technique with local analyses such as computational intensity estimation and memory footprint computation is a challenging issue. The model used by computational intensity estimation could also be refined to take into account comparisons with multi-cores in addition to single-core.

Acknowledgement

This work has been funded by the French ANR and is part of the FREIA project. The authors would like to thank Béatrice Creusillet and Pierre Jouvelot for their valuable comments. We acknowledge the PIPS and Par4All teams for all the accomplished work.

References

1. Alias, C., Darte, A., Plesco, A.: Optimizing DDR-SDRAM communications at C-level for automatically-generated hardware accelerators an experience with the Altera C2H HLS tool. In: ASAP. pp. 329–332 (2010)

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. 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 (DATE'2008). pp. 610–615 (Dec 2008)
4. CAPS Entreprise: HMPP workbench. `http://www.caps-entreprise.com/hmpp.html`
5. Creusillet, B., Irigoin, F.: Interprocedural array region analyses. Int. J. Parallel Program. 24(6), 513–546 (1996)
6. 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)
7. Guo, Z., Najjar, W., Buyukkurt, B.: Efficient hardware code generation for FPGAs. ACM Trans. Archit. Code Optim. 5(1), 1–26 (2008)
8. HPC Project: Par4All initiative. `http://www.par4all.org`
9. Irigoin, F., Jouvelot, P., Triolet, R.: Semantical interprocedural parallelization: An overview of the PIPS project. In: International Conference on Supercomputing, Cologne (june 1991)
10. Irigoin, F., Silber-Chaussumier, F., Keryell, R., Guelton, S.: PIPS Tutorial at PPoPP 2010. `http://pips4u.org/doc/tutorial`
11. Karimi, K., Dickson, N.G., Hamze, F.: A performance comparison of CUDA and OpenCL. CoRR abs/1005.2581 (2010)
12. Khronos OpenCL Working Group: The OpenCL Specification, version 1.1 (september 2010)
13. Kindratenko, V.V., Brunner, R.J., Myers, A.D.: Mitrion-C application development on SGI Altix 350/RC100. In: FCCM. pp. 239–250 (2007)
14. Lee, S., Min, S.J., Eigenmann, R.: OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In: PPoPP '09. pp. 101–110 (2009)
15. Liao, C., Quinlan, D.J., Vuduc, R., Panas, T.: Effective source-to-source outlining to support whole program empirical optimization. In: International Workshop on Languages and Compilers for Parallel Computing (LCPC) (Oct 2009)
16. NVIDIA: NVIDIA CUDA Reference Manual 3.2. `http://www.nvidia.com/object/cuda_develop.html` (Jan 2011)
17. Orfanidis, S.J.: Introduction to signal processing. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1995)
18. Tu, P., Padua, D.A.: Automatic array privatization. In: Compiler Optimizations for Scalable Parallel Systems Languages. pp. 247–284 (2001)
19. Wolfe, M.: Implementing the PGI accelerator model. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU'10). pp. 43–50 (2010), `http://doi.acm.org/10.1145/1735688.1735697`
20. Zhou, L.: Complexity estimation in the PIPS parallel programming environment. In: CONPAR. pp. 845–846 (1992)