Introducing 'Bones': A Parallelizing Source-to-Source Compiler Based on Algorithmic Skeletons

Cedric Nugteren Henk Corporaal
Eindhoven University of Technology, The Netherlands
http://parse.ele.tue.nl/
{c.nugteren, h.corporaal}@tue.nl

ABSTRACT

Recent advances in multi-core and many-core processors requires programmers to exploit an increasing amount of parallelism from their applications. Data parallel languages such as CUDA and OpenCL make it possible to take advantage of such processors, but still require a large amount of effort from programmers.

A number of parallelizing source-to-source compilers have recently been developed to ease programming of multi-core and many-core processors. This work presents and evaluates a number of such tools, focused in particular on C-to-CUDA transformations targeting GPUs. We compare these tools both qualitatively and quantitatively to each other and identify their strengths and weaknesses.

In this paper, we address the weaknesses by presenting a new classification of algorithms. This classification is used in a new source-to-source compiler, which is based on the *algorithmic skeletons* technique. The compiler generates target code based on *skeletons* of parallel structures, which can be seen as parameterisable library implementations for a set of algorithm classes. We furthermore demonstrate that the presented compiler requires little modifications to the original sequential source code, generates readable code for further fine-tuning, and delivers superior performance compared to other tools for a set of 8 image processing kernels.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms

Performance, Languages

Keywords

Parallel Programming, Graphics Processing Units, Algorithmic Skeletons, Source-to-Source Compilation

GPGPU-5, March 03 2012, London, United Kingdom

1. INTRODUCTION

Throughout the past decades, the tremendous growth of single-core processor performance has been the key-enabler for technology to become ubiquitous in our society. This growth ended around 2004, limited by among others power dissipation. Performance growth has been re-enabled recently by adding multiple processor cores per chip. Enabled by Moore's law, this trend (multi-core) is expected to continue for the next decade and is expected to enable 100-core processors by 2020 [6]. Meanwhile, another trend (many-core) already enables more than 500 cores per chip. This is exemplified by the Graphics Processing Unit (GPU). The GPU, driven by the computer gaming industry, bundles numerous smaller processing cores to create a massively parallel and high throughput processor architecture.

Both trends (multi-core and many-core) require application programmers to exploit more and more parallelism. At the same time, these programmers face an increasing amount of heterogeneous compute platforms and processors, as both multi-core and many-cores architectures are used side-by-side in the same device or even on the same chip [12]. In order to fully utilize multi-core and many-core processors, current and future programmers are challenged to achieve a high amount of parallelism and handle heterogeneity.

Recently, many new parallel programming languages such as OpenCL, NVIDIA's CUDA and Intel's TBB have emerged to help application programmers to face the challenges of programming multi-core and many-core processors. Still, even these new languages require the programmers to explicitly extract the parallelism from their applications. They have to create and manage thousands of threads, deal with concurrent execution, synchronization, race conditions and atomicity. Additionally, these languages expose a complete memory hierarchy to the programmer. Programmers have to explicitly copy data between different processors, make use of scratchpad memories, and deal with issues such as memory access patterns and bank conflicts.

In a recent study, programming of current and future processors has been listed as one of the major challenges in computing [12]. Already a significant amount of research has been dedicated to this challenge, but a solution applicable in practice is yet to be found. In this work, we focus on the parallel programming challenge for GPUs, and, in particular, search for a solution which is applicable in practice. We summarize the contributions of this work as follows:

- We compare four existing tools addressing the programmability challenges of GPUs both qualitatively and quantitatively.

- We introduce a new algorithm classification, which can be used among others with the *algorithmic skeletons* technique to address the programmability challenge.
- We introduce a new source-to-source compiler to ease GPU programming, which is based on the algorithmic skeletons technique and the new algorithm classification. We focus heavily on the applicability of this compiler in practice, and address shortcomings of other tools with similar goals.

The remainder of this paper is organized as follows. First, four existing tools are introduced and compared against each other qualitatively in section 2. This section also provides a background on the algorithmic skeletons technique and discusses its shortcomings. Next, in section 3, the new algorithm classification is introduced. This classification is used in the new source-to-source compiler, which is introduced and discussed in section 4. The new compiler and the four earlier introduced tools are compared to each other quantitatively in section 5. Finally, we conclude the work in section 6.

2. EXISTING C-TO-CUDA TOOLS

A significant amount of research has been dedicated to the parallel programming challenge. In particular, for GPUs, different theories, methods, guidelines and tools are available. In this section, we select and introduce four of these existing tools and perform a qualitative comparison, supported by table 1. In section 5, we also make a performance-wise comparison. To the best of our knowledge, this work is the first direct comparison between such tools.

To address the programmability issues of GPUs, several source-to-source compilers which generate GPU code were developed. We focus in particular on source-to-source compilers which use C-code as an input and output CUDA. In this section we identify tools based on three different methods: 1) directive based tools, 2) tools using algorithmic skeletons, and 3), polyhedral model based tools. We discuss a directive based tool (*hi*CUDA [10]), an algorithmic skeleton tool (SkePU [9]), and a tool based on the polyhedral model (Par4All [1]) in more detail. Additionally, we discuss a commercial tool: PGI Accelerator [23].

In order to illustrate the programming style for these tools and the required programming effort, we take C-code for a convolution operation as an example. The computational part of the algorithm is shown in listing 1.

```
1 int N = 512*512;
2 for(i=1;i<N-1;i=i+1)
3   B[i] = 3*A[i-1] + 4*A[i] + 3*A[i+1];
```

Listing 1: Convolution C-code example.

2.1 Directives using *hi*CUDA

Source-to-source compilers based on directives rely on the help of programmers to generate code. A common case is the use of annotations (e.g. pragmas) in the source code, which guide (or: direct) the compiler towards generating efficient target code. We identify a number of C-to-CUDA compilers using such a technique, most of them based on OpenMP-style pragmas [8] [13] [14]. Furthermore, we identify the domain specific tool Mint [22] and the CUDA-to-CUDA optimizer CUDA-lite [21]. In this work, we choose to discuss the C-to-CUDA compiler *hi*CUDA [10] in more detail, partly because of the availability of source-code and partly because of the relatively high number of scientific citations.

To illustrate the use of *hi*CUDA, we apply the directives to the convolution example. As seen in listing 2, the original code is still present in the *hi*CUDA code (lines 1, 7 and 10), but a number of directives have been added. First of all, array sizes have to be set in case of dynamically allocated memory (line 2). Secondly, memory has to be managed, including allocation on the GPU (lines 3-4), copying between CPU and GPU (lines 3 and 13), and freeing on the GPU (line 14). Finally, the kernel has to be defined (lines 5-12). Most of these directives translate directly to CUDA statements.

```
1  int N = 512*512;
2  #pragma hicuda shape A[N] B[N]
3  #pragma hicuda global alloc A[*] copyin
4  #pragma hicuda global alloc B[*]
5  #pragma hicuda kernel conv tblock(N/256)
       thread(256)
6  #pragma hicuda loop_partition over_tblock
       over_thread
7  for(i=1;i<N-1;i=i+1) {
8    #pragma hicuda shared alloc A[i-1:i+1]
         copyin
9    #pragma hicuda barrier
10   B[i] = 3*A[i-1] + 4*A[i] + 3*A[i+1];
11 }
12 #pragma hicuda kernel_end
13 #pragma hicuda global copyout B[*]
14 #pragma hicuda global free A B
```

Listing 2: Convolution using *hi*CUDA directives.

Using *hi*CUDA, the programmer is still required to have GPU programming experience. For example, the programmer has to specify the desired amount of threads and threadblocks and has to specify which for-loops to parallelize. Additionally, when using the on-chip scratchpad memory, the programmer has to supply directives defining which memory section to store locally and when to synchronize between threads (lines 8-9 in listing 2). An advantage of *hi*CUDA is its interprocedural support. For example, 'kernel' directives can be placed around function calls, while others can be placed within these functions. The code which *hi*CUDA generates can be inspected and modified. However, the generated code is not straightforward to read and thus not suitable for further fine-tuning.

2.2 Algorithmic skeletons using SkePU

Algorithmic skeletons were introduced in 1991 by Cole [7], a technique which revolves around a set of parameterisable *skeleton implementations*. Each skeleton implementation can be seen as template code for a specific algorithm class on a target architecture. Programmers are able to generate efficient target code by first identifying a number of lines of code as a certain class, followed by invoking the corresponding skeleton implementation. If no skeleton implementation is available for the specific class-architecture combination, it can be added manually. Future algorithms of the same class can then benefit from re-use of the skeleton code.

Applying algorithmic skeletons to many-core architectures such as a GPU has been accomplished recently in several

Table 1: A comparison of properties of four existing tools and the tool introduced in this work (Bones). We annotate properties either with a minus-sign (negative property) or with a plus-sign (positive property).

	*hi*CUDA	SkePU	PGI Accelerator	Par4All	Bones
tested version	0.9	0.7	11.10	1.2	0.8
availability	open-source	open-source	commercial	open-source	open-source
target language support	CUDA	CUDA/OpenCL	CUDA	CUDA/OpenMP[1]	CUDA/OpenCL
compilable without the tool?	yes (+)	no (-)	yes (+)	yes (+)	yes (+)
uses regular C data types?	yes (+)	no (-)	yes (+)	no malloc (-)[1]	yes (+)
array dimensions are required at?	run-time (+)	run-time (+)	compile-time (-)	run-time (+)	compile-time (-)
readability of the code	moderate (-)	N/A (-)	unreadable (-)	good (+)	good (+)
has multi-GPU support?	no (-)	yes (+)	no (-)	no (-)	no (-)
has support for kernel fusion?	no (-)	no (-)	no (-)	no (-)[1]	no (-)

works [9] [16] [18] [20]. An implementation is only made publicly available for SkePU [9], which supports both CUDA and OpenCL as targets. In this work, we focus on SkePU.

We illustrate the use of SkePU with the convolution example as shown in listing 1. The SkePU source code in listing 3 is obtained by performing a number of steps on the convolution code. First of all, we need to select a matching skeleton for our code. SkePU currently supports 7 skeletons (map, mapArray, mapOverlap, mapReduce, reduce, scan and generate), from which we select mapOverlap to match the convolution code (line 13). Secondly, we copy and convert our arrays into the `skepu::Vector` or `skepu::Matrix` containers (lines 8-12 and 15-17). Finally, we define the functionality (lines 2-4) and invoke the convolution kernel (line 14).

```
1  // Global function definition
2  OVERLAP_FUNC(conv, float, 1, in,
3      return 3*in[-1] + 4*in[0] + 3*in[1];
4  )
5
6  // Main function
7  int N = 512*512;
8  skepu::Vector<float> A_v(N);
9  skepu::Vector<float> B_v(N);
10 for (i=0;i<N;i=i+1) {
11   A_v[i] = A[i];
12 }
13 skepu::MapOverlap<conv> convolve(new conv);
14 convolve(A_v,B_v);
15 for (i=0;i<N;i=i+1) {
16   B[i] = B_v[i];
17 }
```

Listing 3: Convolution using SkePU's skeletons.

SkePU does not require a separate compilation stage, instead, including the supplied header files and compiling with NVIDIA's CUDA compiler *nvcc* is sufficient. However, using SkePU requires a rewrite of the original code as shown in listing 3, which can be disadvantageous. Also, the tool requires data structures to be in a special format. In some cases, this will not pose a problem, as the whole program can be designed using the SkePU containers (see the example in listing 3). In other cases, when kernels are considered as small parts of an existing larger application, a copy-in and a copy-out is required. We consider the latter case in this work, but acknowledge that this restriction will not always pose a problem. We furthermore note that SkePU has support for lazy memory copying, can generate multi-GPU code, and supports multiple targets (OpenCL and CUDA).

2.3 PGI Accelerator

PGI Accelerator [23] is a commercial C/Fortran-to-CUDA source-to-source compiler. It performs extensive code analysis, but also relies on a number of programmer supplied directives. We illustrate the use of the compiler by showing the convolution example in listing 4. In the best-case, the PGI Accelerator requires only one directive (line 2) and will analyze which arrays to copy to and from the GPU, which loops to parallelize, which temporary results to store in on-chip memory, etc. If the PGI Accelerator cannot find these details, the user will be asked to supply directives. In this case, we provide a directive (line 4) to inform the compiler that the loop iterations are independent of each other.

```
1 int N = 512*512;
2 #pragma acc region
3 {
4   #pragma acc for independent
5   for(i=1;i<N-1;i=i+1)
6     B[i] = 3*A[i-1] + 4*A[i] + 3*A[i+1];
7 }
```

Listing 4: Convolution using the PGI Accelerator.

The PGI Accelerator source-to-source compiler provides information to the user as to how the CUDA implementation is generated. The programmer can then supply additional directives to guide the compiler in a specific direction. The compiler furthermore lists details such as occupancy, type of memory accesses, register usage, etc.

2.4 Using the polyhedral model with Par4All

Lastly, we briefly discuss C-to-CUDA compilers based on the polyhedral model. Such compilers are able to perform dependency analysis and loop transformations for affine loop structures. Recently, support for GPUs has been added to these compilers. Compilers based on the polyhedral model are promising in the sense that they require little or no input from the programmer, but they might be limited by restricting transformations only to affine loop structures.

We identify three efforts to extend existing polyhedral compilers in order to support C-to-CUDA. One of these extends the polyhedral compiler collection (PoCC) to support C-to-CUDA compilation [2]. Although this work is promis-

[1] A to be released version of Par4All (1.3.1) will improve on these aspects. This new version adds OpenCL as a target language, does support malloc-statements, and provides an algorithm to enable kernel fusion.

ing, a compiler has not been made publicly available yet. In other work Pluto [3] is introduced, which is an automatic parallelizing compiler based on the polyhedral model. It is not further discussed in this work, since support for CUDA is currently limited: it only generates the CUDA kernel code, the CUDA host code has to be written manually and integrated in the original program by hand. Finally, we identify the Par4All project [1], which we discuss in more detail.

The code transformation and parallelization framework PIPS is the main component of the Par4All project. In this project, a source-to-source compiler is developed based on PIPS, which targets the generation of among others CUDA code. The Par4All compiler is fully-automatic, taking unmodified C-code as input (e.g. the code in listing 1). Still, a number of restrictions apply and code restructuring might be required to achieve good performance. The generated target code contains a large amount of macro's, which hide CUDA statements from the source file. The generated code remains therefore readable.

2.5 Addressing the applicability of existing C-to-CUDA tools

As stated before, we focus in this work on the applicability of the discussed tools in practice. Although these tools all greatly reduce the amount of effort needed to program a GPU, most programmers still use a native programming environment (e.g. CUDA). A major reason for this is performance: although some tools yield performance close to hand written code in some cases, they, as we will see later, perform more than 10x worse in other cases. We believe that programmers are willing to spend more effort to obtain significantly improved performance for these cases.

Algorithmic skeletons provide a way to generate high performance code, while using little or no code analysis nor transformation techniques. Skeletons can be seen as highly optimized library implementations for classes of algorithms instead of for individual algorithms. The only question remains is how to define such classes and their corresponding skeletons. In this work, we aim to increase the applicability of the algorithmic skeletons technique in practice. To do so, we ideally require an algorithm classification which is complete, detailed and straightforward to understand. The granularity of such a classification is of high importance for the applicability. When using algorithmic skeletons, a finer-grained classification could yield a higher performance. On the other hand, if the classification is coarser-grained, it can be easier to use and to understand, and reduces the required number of skeleton implementations. In this work, we find a solution to this trade-off by introducing a modular and parameterisable classification. This enables a fine-grained classification while using a limited vocabulary.

Next to a new algorithm classification, we aim to improve upon existing C-to-CUDA tools by the following two points. Firstly, we aim to generate target code which is both editable and readable by the programmer. In this way, code can be fine-tuned manually to achieve maximal performance. Secondly, we want to keep the source code close to the original code. In this way, less effort from the programmer is required and code can still be compiled using a normal C-compiler.

3. ALGORITHM CLASSIFICATION

To support a new source-to-source compiler based on algorithmic skeletons, we introduce a new algorithm classification. Detailed information on the classification can be found in an extended technical report [15]. In this section, we first give a small toy example, after which we introduce the vocabulary and grammar of the classification. We illustrate this with a number of example classes and their formal notations. Following, from the field of image processing, we classify a number of elementary algorithms using the introduced classification. Finally, we evaluate the classification. The following notations are used in this section: (1) X^i, Y^i and Z^i represent i-dimensional data structures, and (2) $X^i_{\mathbf{d}}$ denotes the contents of a data structure X^i at coordinate $\mathbf{d}$.

3.1 Code examples

We introduce the classification by giving an intuitive feel through three example code snippets. The examples as given in listing 5 are classified as follows:

- In lines 1-2 a vector of size K is element-wise multiplied, incremented, and stored as another vector. Since every *element* of the input corresponds to an *element* of the output and the vector size is K, we classify this code snippet as 'K|element → K|element'.
- The for-loop in lines 4-5 performs a similar operation, but now also requires two *neighbours* to compute one output *element*. The classification becomes 'K|neighbourhood(3) → K|element', since the neighbourhood is of size 3 (including the element itself).
- The final snippet (lines 7-8) processes the input per *element*, but stores the result in a *shared* output. It is therefore classified as 'K|element → 1|shared', with 1 being the size of the output.

```
1  for(i=0;i<K;i=i+1)
2      B[i] = 2 * A[i] + 5;
3
4  for(i=0;i<K;i=i+1)
5      B[i] = 3*A[i-1] + 4*A[i] + 3*A[i+1];
6
7  for(i=0;i<K;i=i+1)
8      B = B + A[i];
```

Listing 5: Three example code snippets of different classes.

3.2 Vocabulary and grammar

The classification is generalized and described using a vocabulary and a grammar. The classification's grammar is defined as

$$P\ S|D\ [\wedge\ S|D]^*\ \rightarrow\ S|D\ [\wedge\ S|D]^*$$

in which the asterisk symbol (*) implies zero or more occurrences of everything contained by brackets preceding the asterisk. In this definition, P represents a prefix and occurrences of $S|D$ represent input and output data-structures. The prefix P, data access patterns D and dimensions S use the following vocabulary:

$$P = \begin{pmatrix}\text{unordered}\\ \text{multiple}\end{pmatrix} \qquad S = \begin{pmatrix}\text{A}\\ \text{AxB}\\ \text{...}\\ \text{Ax...xN}\end{pmatrix}$$

Table 2: A number of example classes illustrate the algorithm classification. The formal notation defines the properties of a class.

id	classification	formal notation
1	AxB|element → AxB|element	$\forall \mathbf{d} \in \mathcal{D} : f\left(X^2_{\mathbf{d}}\right) \rightarrow Y^2_{\mathbf{d}}$
2	unordered AxB|element → AxB|element	$\forall \mathbf{d} \in \mathcal{D}\ \exists \mathbf{d}' \in \mathcal{D} : f\left(X^2_{\mathbf{d}}\right) \rightarrow Y^2_{\mathbf{d}'}$
3	AxB|tile(1xB) → A|element	$\forall \mathbf{d} \in \mathcal{D} \mid d = k \cdot \mathcal{T}' : f\left(\forall t' \in \mathcal{T}'\left(X^2_{\mathbf{d}+t'}\right)\right) \rightarrow Y^2_c \mid c = d_x$
4	AxB|tile(UxV) → $\frac{A}{U}$x$\frac{B}{V}$|element	$\forall \mathbf{d} \in \mathcal{D} \mid d_x = k \cdot U \wedge d_y = k \cdot V : f\left(\forall \mathbf{t} \in \mathcal{T}\left(X^2_{\mathbf{d}+\mathbf{t}}\right)\right) \rightarrow Y^2_{\mathbf{c}} \mid \mathbf{c} = (\frac{d_x}{U}, \frac{d_y}{V})$
5	AxB|tile(UxV) → AxB|tile(UxV)	$\forall \mathbf{d} \in \mathcal{D} \mid d_x = k \cdot U \wedge d_y = k \cdot V : f\left(\forall \mathbf{t} \in \mathcal{T}\left(X^2_{\mathbf{d}+\mathbf{t}}\right)\right) \rightarrow \forall \mathbf{t} \in \mathcal{T}\left(X^2_{\mathbf{d}+\mathbf{t}}\right)$
6	AxB|element → A·UxB·V|tile(UxV)	$\forall \mathbf{d} \in \mathcal{D} : f\left(X^2_{\mathbf{d}}\right) \rightarrow \forall \mathbf{t} \in \mathcal{T}\left(X^2_{\mathbf{c}+\mathbf{t}} \mid \mathbf{c} = (d_x \cdot U, d_y \cdot V)\right)$
7	AxB|neighbourhood(NxM) → AxB|element	$\forall \mathbf{d} \in \mathcal{D} : f\left(\forall \mathbf{n} \in \mathcal{N}\left(X^2_{\mathbf{d}+\mathbf{n}}\right)\right) \rightarrow Y^2_{\mathbf{d}}$
8	AxB|neighbourhood(N) → AxB|element	$\forall \mathbf{d} \in \mathcal{D} : f\left(\forall \mathbf{n}' \in \mathcal{N}'\left(X^2_{\mathbf{d}+\mathbf{n}'}\right)\right) \rightarrow Y^2_{\mathbf{d}}$
9	AxB|element → 1|shared	$\forall \mathbf{d} \in \mathcal{D} : f\left(X^2_{\mathbf{d}}, Y^0\right) \rightarrow Y^0$
10	AxB|element → C|shared	$\forall \mathbf{d} \in \mathcal{D}\ \exists v \in \mathcal{V} : f\left(X^2_{\mathbf{d}}, Y^1_v\right) \rightarrow Y^1_v$
11	AxB|element ∧ AxB|element → AxB|element	$\forall \mathbf{d} \in \mathcal{D} : f\left(X^2_{\mathbf{d}}, Y^2_{\mathbf{d}}\right) \rightarrow Z^2_{\mathbf{d}}$
Natural numbers k		$k \in \mathbb{N}$
Data structure coordinates $\mathcal{D}$ — AxB		$\mathcal{D} = \{(x, y) \mid 0 \leq x < A \wedge 0 \leq y < B\}$
Tile coordinates $\mathcal{T}$ — UxV		$\mathcal{T} = \{(x, y) \mid 0 \leq x < U \wedge 0 \leq y < V\}$
Tile coordinates $\mathcal{T}'$ — B		$\mathcal{T}' = \{(x, y) \mid 0 \leq x < B \wedge y = 0\}$
Neighbourhood coordinates $\mathcal{N}$ — NxM		$\mathcal{N} = \{(x, y) \mid -\lfloor\frac{N-1}{2}\rfloor \leq x \leq \lceil\frac{N-1}{2}\rceil \wedge -\lfloor\frac{M-1}{2}\rfloor \leq y \leq \lceil\frac{M-1}{2}\rceil\}$
Neighbourhood coordinates $\mathcal{N}'$ — N		$\mathcal{N}' = \{(x, y) \mid -\lfloor\frac{N-1}{2}\rfloor \leq x \leq \lceil\frac{N-1}{2}\rceil \wedge y = 0\}$
Vector index $\mathcal{V}$ — C		$\mathcal{V} = \{(x) \mid 0 \leq x < C\}$

$$D = \begin{pmatrix} \text{element} \\ \text{tile}\,(S) \\ \text{neighbourhood}\,(S) \\ \text{shared} \end{pmatrix}$$

The classification introduces the operation $f()$, which is defined as the mathematical computation required to be performed per input data element. The different data access patterns D are defined as follows:

- The *element* access pattern represents a single independent access of a data structure's contents at each coordinate.
- When accessing data in a *tile(T)* pattern, a structure of dimensions T is accessed simultaneously, but independent of other tiles in the same data structure. There is no data re-use, all contents are accessed once.
- The *neighbourhood(N)* access pattern is similar to the *element* pattern, but enables overlap. The pattern describes the re-use of a data structure's contents for a neighbourhood of dimensions N centered around each coordinate of the structure. This data access pattern is to be used as input pattern only.
- The *shared* data access pattern is an output pattern only. It is used when multiple accesses occur to contents at one coordinate in a data structure, and can also involve reading from the output.

In general, the relation between the input and output patterns is based on a one to one coordinate mapping. The use of the *unordered* prefix implies that any coordinate mapping can be used, such as a scattered or reverse mapping. The *multiple* prefix can be used in combination with the shared pattern, when more than one output is written by the operator $f()$.

Table 3: Example primitives and their corresponding class-id. These id's relate to the classes as defined in table 2.

example primitive	class-id
image copy	1
binarization	1
colour translation	1
gamma correction	1
contrast enhancement	1
arithmetic & logic (monadic)	1
rotate 90 degrees	2
xy-mirroring	2
rotate (forward warping, no interpolation)	2
shear-x (forward warping, no interpolation)	2
column projection	3
pixelization	4 followed by 6
adaptive binarization (alternative 1)	4 followed by 6
2D type II discrete cosine transform	5
convolution (static filter)	7
adaptive binarization (alternative 2)	7
erode	7
dilate	7
separable 2D filter (in dimension A)	8
separable 2D filter (in dimension B)	8
sum	9
min/max	9
histogram	10
differencing	11
arithmetic & logic (dyadic)	11

3.3 Example classes and example primitives

To illustrate the use of the classification and to get an intuitive feel for it, we present a number of example classes in table 2 (more examples and a detailed explanation are found in [15]). We refer to the *id*'s listed in this table as

a reference to the corresponding class and its formal notation. In this table, we see two embarrassingly parallel classes (class 1 and 2), of which the second class' coordinate relation is scattered. Classes 3, 4, 5, and 6 are tile-based computations. While class 6's output has increased dimensions, class 3 and 4's output is smaller compared to the input. For class 5, dimensions remain unchanged. Following, two neighbourhood-based computations are listed, using either a 2D neighbourhood (class 7) or a 1D neighbourhood (class 8). Classes 9 and 10 perform a scalar and a vector-based reduction respectively. Finally, class 11 uses multiple inputs.

Related to the example classes in table 2, we list a number of elementary algorithms (*primitives* or *kernels*) in table 3. These primitives can be seen as building blocks for larger algorithms and applications in the domains of image processing and computer vision. We take four primitives from table 3 to illustrate a number of classes from table 2:

- We classify `contrast enhancement` as 'AxB|element → AxB|element', in which A and B are the x and y-dimensions of the input and output image. The structure of the primitive resembles the example given in lines 1-2 of listing 5, but performs a different operation using a different function $f()$.
- The primitive `rotate 90 degrees` is classified as 'unordered AxB|element → AxB|element'. As before, an element-wise operation is performed, but now with a scattered coordinate mapping, which is necessary to perform the rotation.
- The algorithm `pixelization`, consisting of two smaller parts, is classified as a sequence of 'AxB|tile(UxV) → $\frac{A}{U}$x$\frac{B}{V}$|element' and '$\frac{A}{U}$x$\frac{B}{V}$|element → AxB|tile(UxV)'. First, an operation is performed per *tile* of the input image, resulting in a fewer number of *elements* in an intermediate image. Then, each of these intermediate *elements* is copied into a *tile* of the output image.
- Many 2D convolution filters can be separated into a sequence of two 1D filters. We classify one of these `separable 2D filters` as a 1D neighbourhood operation on a 2D input image: 'AxB|neighbourhood(N) → AxB|element'.

3.4 Evaluation of the classification

The presented classification is evaluated in detail in [15]. In this section, we evaluate the classification with respect to completeness, granularity and understandability and briefly compare the classification to existing work.

Completeness We have shown that a significant amount of algorithm primitives can be classified under the presented classification. Larger algorithms can be split into multiple sections, each with code sections corresponding to a primitive. Irregular algorithms and primitives with limited or no parallelism can be classified under less constrained classes, such as the unrestricted class 'S|tile(S) → S|shared'. Furthermore, the use of a modular classification enables the support for any number of input and output data structures.

Granularity We add parameters for data structure, neighbourhood and tile sizes, making it possible to achieve a fine granularity if needed.

Understandability The classification uses a limited set of data-structure access patterns, suggesting ease of adaptability.

The presented classification shows many similarities to the domain specific classifications presented in [4], [9] and [19], in which element-wise, neighbourhood and reduction classes also exist. Two major differences are observed: (1) the modularity and (2) the parameter-based approach. Firstly, with a modular classification, classes do not have to be defined, but can be constructed using a given grammar and a limited vocabulary. Secondly, the use of parameters provides a manner to distinguish different sizes and dimensions from each other - if needed. For example, we can distinguish a large neighbourhood from a small neighbourhood (and thus create two skeletons), but we can group together 1D and 2D tiles (creating one skeleton).

In comparison to each of the 10 classifications discussed in [5], we propose significantly more different classes (modularity and parameters) and provide a much more detailed description of the primitive (parameters). The more detailed a classification is, the more specific skeletons can be constructed. If skeletons are very specific, the resulting code will benefit the most from the target processor's capabilities.

4. SKELETONS REVISITED

In this work we present Bones, a source-to-source compiler based on algorithmic skeletons and the presented algorithm classification. The compiler takes C-code annotated with class information as input and generates parallelized target code. At this moment, targets include NVIDIA GPUs (through CUDA), AMD GPUs (through OpenCL) and x86 CPUs (through OpenCL).

To illustrate the use of Bones, we evaluate the example convolution code as shown in listing 1 (section 2). Bones requires the programmer to supply a directive to specify the algorithm class (line 2 in listing 6). A second directive denotes the end of the kernel, which programmers also can use to provide an optional name (line 5). This section explains the internals of the compiler, and compares the compiler qualitatively with the earlier introduced tools.

```
1 int N = 512*512;
2 #pragma kernel N|neighb(3) -> N|element
3 for(i=1;i<N-1;i=i+1)
4   B[i] = 3*A[i-1] + 4*A[i] + 3*A[i+1];
5 #pragma endkernel conv
```

Listing 6: Convolution code accelerated with Bones.

4.1 The Bones source-to-source compiler

Bones is written in the Ruby programming language and is available through our website[2]. The compiler is based on the C-parser *CAST*[3], which is used to parse the input code into an abstract syntax tree (AST) and to generate the target code from a transformed AST. A high-level overview of Bones is shown in figure 1, while a more detailed view is shown in figure 2.

[2]http://parse.ele.tue.nl/
[3]http://cast.rubyforge.org/

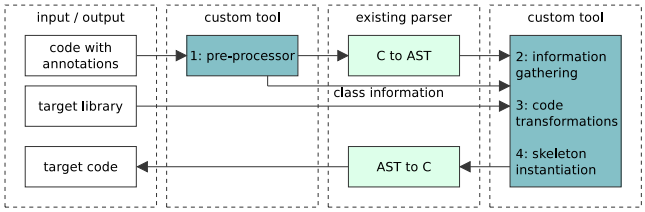

Figure 1: High level overview of the Bones source-to-source compiler.

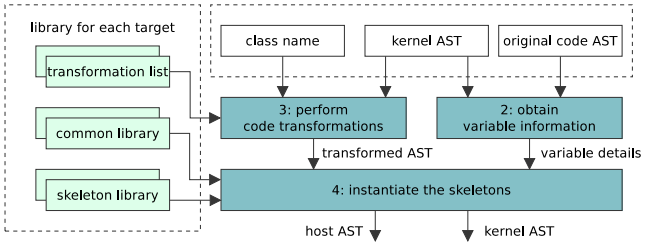

Figure 2: Detailed view of the core components of Bones.

The Bones compiler performs four steps. Firstly, a preprocessor extracts the user supplied class information from the source code (label 1 in figure 1). Code in between the `kernel` and `endkernel` statements is defined as the kernel code. Secondly, the AST of the kernel code and the AST of the original code are analyzed to obtain information on the used variables (label 2 in figures 1 and 2), e.g. are they dynamically or statically allocated, are they private to the kernel, of which dimension are they. Thirdly, the kernel AST is transformed according to a per class unique transformation list (label 3 in figures 1 and 2). These transformations are fairly basic, e.g. renaming of variables, removal of the outer loop. Lastly, the skeletons are instantiated (label 4 in figures 1 and 2). A common library supplies parameterized host code, including memory allocation and memory transfer mechanisms. The skeleton library supplies both parameterized host code and kernel code. The parameters are instantiated based on the extracted variable information and the user supplied class details.

Bones uses the in this work introduced algorithm classification as a basis for the skeleton library. Currently, a total of 24 skeletons are provided for three different targets: `gpu-nvidia`, `gpu-amd` and `cpu-opencl`. For the latter two, 6 skeletons are implemented in OpenCL (different versions for different targets), while 12 skeletons are already available in CUDA for the `gpu-nvidia` target, supporting at least all the primitives listed in table 3.

4.2 Evaluating Bones

The source-to-source compiler Bones is evaluated qualitatively in this section, supported by table 1. The presented compiler improves upon existing C-to-CUDA tools by three main factors:

1. Bones generates readable and editable code. Because the code generated was kept in AST form, it remains very readable. The original variable names and coding style are still present in the generated code. Furthermore, the skeletons are formatted to contain relevant variable names and comments. Readability is paramount for further fine-tuning of the algorithm in case the provided skeletons are not fine-grained enough. The discussed tools *hi*CUDA and PGI Accelerator do provide editable code, but the readability is presenting a major hurdle. SkePU instantiates skeletons at runtime, leaving no possibilities for further fine-tuning. Par4All in contrast does generate readable code.

2. Bones is based on a well-defined algorithm classification, which is presented in this work. In SkePU for example, a new skeleton with a new name might be added by the developers. In the case of Bones, the classification's grammar and vocabulary is already defined. Additionally, the classification is much finer-grained, providing data, tile and neighbourhood dimensions and sizes.

3. Similar to PGI Accelerator, only a minimum effort is required (two directives) to transform the original code into code suitable for the compiler. Additionally, the original code is kept intact, which can be useful to test functional correctness or to execute on legacy systems. The tools *hi*CUDA, PGI Accelerator and Par4All also have this latter advantage, but SkePU in contrast requires a complete rewrite of the code.

Bones is also known to have several weaknesses. Firstly, as with other skeleton-based approaches, Bones relies on the programmer to identify and select a corresponding class for each primitive. If the programmer selects a non-matching class, the compiler will generate incorrect code. Secondly, Bones accepts only a subset of C for the kernel code. For example, function calls are not supported as of now. Thirdly, reflected by table 1, we note that Bones currently does not allow array sizes to be defined at run-time, does not support multi-GPU and is not able to fuse multiple kernels. These weaknesses will be addressed in future work.

5. PERFORMANCE EVALUATION

In this work, we discussed four existing tools to ease GPU programming and compared them qualitatively. Also, we introduced a new source-to-source compiler based on algorithmic skeletons. In this section, we evaluate all five tools performance-wise, which we base on a real-life example application containing a variety of kernels. Additionally, we evaluate Bones for different targets.

In our setup, we use a system consisting of a 4-core Intel Core-i7 930 CPU and two GPUs: an NVIDIA GeForce GTX 470 and an AMD Radeon HD5850. Both the CPU and the AMD GPU are programmed using OpenCL through AMD's APP version 2.5. The NVIDIA GPU is programmed using CUDA SDK version 4.1.

5.1 An example application

To compare the existing tools and the new source-to-source compiler Bones, we select an example case-study application. We use a computer vision application with a variety of image processing primitives. This application is used in the production process of organic LEDs, where the centers of individual LEDs have to be identified under challenging throughput and latency requirements. This can be achieved using the 3-stage *fast focus on structures* flow, which is discussed in detail in [11]. The three stages are applied subse-

Table 4: Classification of the application's image processing primitives according to the presented grammar and vocabulary, and a comparison of the required code changes/additions for four tools in number of lines of code (including directives).

	primitive	classification	*hi*CUDA	SkePU	PGI Accelerator	Par4All	Bones
histo	histogram	1024x1024|element → 256|shared	29 LoC	-	17 LoC	0 LoC	2 LoC
max-1	maximum-1	256|element → 1|shared	-	7 LoC	4 LoC	-	2 LoC
max-2	maximum-2	2097152|element → 1|shared	-	7 LoC	4 LoC	-	2 LoC
thres	threshold	1024x1024|element → 1024x1024|element	11 LoC	15 LoC	3 LoC	0 LoC	2 LoC
eros1	erosion 7x7	1024x1024|neighb(7x7) → 1024x1024|element	12 LoC	21 LoC	4 LoC	0 LoC	2 LoC
eros2	erosion 1D	2097152|neighb(7) → 2097152|element	10 LoC	15 LoC	3 LoC	0 LoC	2 LoC
xproj	X-projection	1024x1024|tile(1x1024) → 1024|element	8 LoC	-	2 LoC	0 LoC	2 LoC
yproj	Y-projection	1024x1024|tile(1024x1) → 1024|element	8 LoC	-	2 LoC	0 LoC	2 LoC

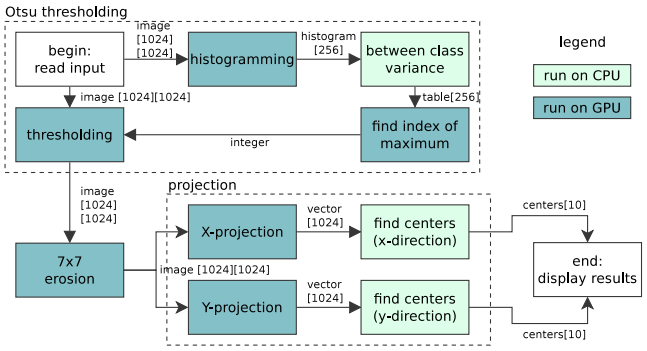

Figure 3: The fast focus on structures flow, using a 1024x1024 pixel input image with a 10x10 grid of LED-structures present.

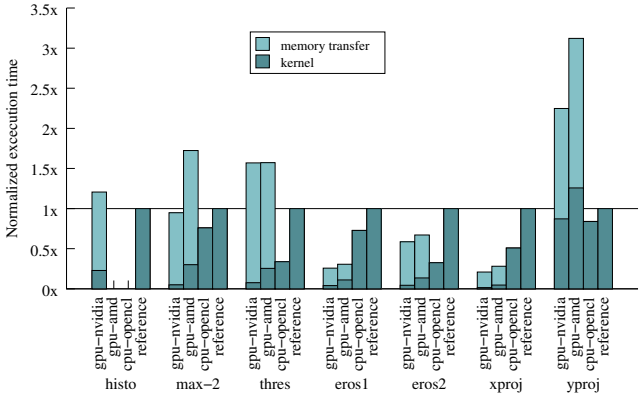

Figure 4: Normalized execution time of the 8 primitives with respect to the single-threaded reference C-code. Both the kernel execution time and the memory transfer time are measured.

quently: (1) Otsu thresholding, (2) erosion, and (3) projection. A flow chart is shown in figure 3, in which the individual image processing primitives are shown. In this figure, 6 primitives are considered targets for GPU acceleration.

The fast focus on structures application is slightly modified in order to create more class diversity in two ways: 1) a 1D erosion kernel applied to a 1D input array is added, and 2), a second maximum primitive is introduced, which works on a significantly larger array compared to the original. The new total of 8 primitives found in the extended fast focus on structures application are classified as shown in table 4.

5.2 Comparing different targets for Bones

Bones supports three different targets. In this section we present the performance numbers for the three targets compared to single-threaded non-optimized reference C-code. This comparison is made to demonstrate the benefits of accelerating code using OpenCL or CUDA. The reference code is annotated with directives determining the class of each of the primitives. The same annotated code is used to generate code for three different targets using Bones: `gpu-nvidia` (executed on the GTX470), `gpu-amd` (executed on the HD5850) and `cpu-opencl` (executed on the Core-i7). The resulting performance numbers are given in figure 4. In this figure, we do not show timing results for the `max-1` primitive. Its execution time is very small, resulting in normalized execution times higher than 10x for the OpenCL and CUDA implementations due to the kernel launch time overheads. Also, no results are shown for the `histo` primitive for the `gpu-amd` and `cpu-opencl` targets. This is because no skeleton has been implemented yet for the corresponding class.

We observe from figure 4 that accelerating using a GPU yields a significant performance increase for most of the cases, especially when considering only the kernel execution times. Even without using additional hardware, the OpenCL code running on the CPU provides a performance increase over the reference C implementation in all cases, benefiting from multi-threading.

5.3 Bones compared to others

To give an idea how Bones behaves compared to other C-to-CUDA tools, we compare the performance of *hi*CUDA, SkePU, PGI Accelerator and Par4All with Bones for the 8 primitives. The tested code is thoroughly optimized, where necessary with help of the tool's support team.

Firstly, we show the total normalized execution time of the 8 primitives for the five different tools in figure 5. In this figure, we divide the total execution time in three parts: 1) GPU kernel execution time, 2) CPU-GPU and GPU-CPU memory transfer time, and 3) other, including overheads for using the different tools. Secondly, we show an enlarged view of the GPU kernel execution time in figure 6. Finally, we present in table 4 the number of lines of code (LoC) that were changed or added to the primitives to achieve these measurements. Before discussing the results, we make a few remarks regarding figures 5-6 and table 4:

- The tested real-life application is chosen to contain a high diversity of image processing primitives. How-

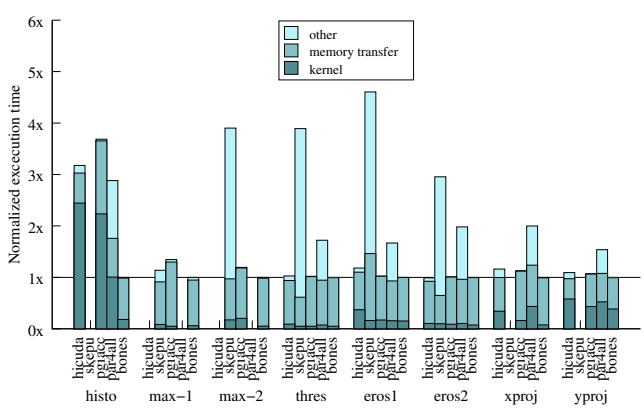

Figure 5: Normalized execution time of the 8 primitives with respect to Bones. All measurements are performed on the GTX470 GPU using CUDA 4.1.

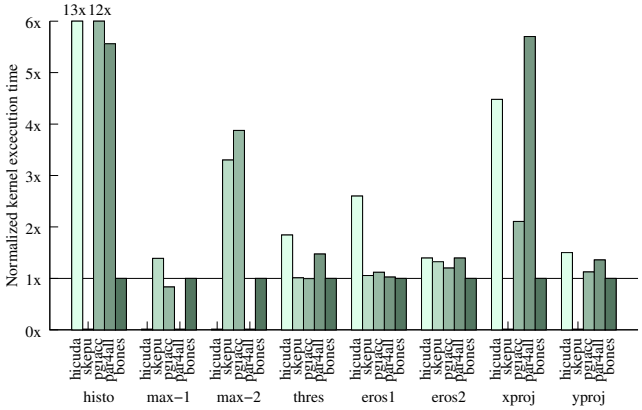

Figure 6: Normalized execution time of the 8 GPU kernels on the GTX470 GPU with respect to Bones. This figure is the kernel-only version of figure 5.

ever, this does not mean that the presented results in this work are necessarily representative for other applications.

- The amount of required code changes in table 4 is merely presented as an indication of the required effort: the information density per line might vary.
- We consider the performance of the kernel itself as the most important metric to compare. The performance of the memory transfers might be relevant for individual kernels, but might be less relevant for larger applications. In some applications with multiple sequential GPU kernels, intermediate memory copies can be omitted.

From the results as shown in figures 5-6 and table 4, we observe the following:

- In a few cases no result could be obtained for *hi*CUDA, SkePU and Par4All. Currently, *hi*CUDA does not support reduction operations, whereas Par4All has limited support for such operations but is unable to generate code for the `max-1` and `max-2` primitives. SkePU does not provide a skeleton for primitives such as `histo` and `yproj`. Furthermore, SkePU does not support non-separable 2D filters, such as `eros1`. In this case however, we do assume the filter to be separable in order to obtain a performance measurement.
- *hi*CUDA and PGI Accelerator show an increased kernel execution time for the `histo` primitive. Due to the inter-loop dependencies of the algorithm, the C-code had to be rewritten for these tools. The execution time is one order of magnitude higher compared to Bones's skeleton implementation, which uses a generalized version of the algorithm presented in [17].
- SkePU shows a large 'other'-bar in figure 5. This is due to the CPU-CPU memory copies required to copy input and output data to and from the SkePU containers. This overhead is non-existent if an application is written entirely using these containers.

From figure 6 we can see that Bones achieves the best GPU kernel performance in most cases. This is no surprise, as the skeletons are heavily optimized for the specific classes. We also see that other tools perform reasonably well, especially in the basic cases such as `thres` and `eros2`. However, in the more complicated cases such as `max-2` and `xproj`, the other tools lose a factor 2-4 compared to Bones. In the case of the `xproj` algorithm for *hi*CUDA, we acknowledge that execution time can be reduced by limiting the amount of uncoalesced accesses. Since this involves manual loop tiling and the creation of a local data copy, we have chosen not to include this in our evaluation.

If we compare the performance for the primitives including memory transfer (figure 5), we see a smaller difference between the different tools. In most cases, memory transfer is the bottleneck. By chaining multiple primitives together, as done in the fast focus on structures application, intermediate memory copies can be omitted. SkePU supports lazy copying, which automatically reduces the number of memory transfers. Similar, Par4All uses an optimization algorithm [1] to reduce CPU-GPU communication. In the case of *hi*CUDA, reducing memory copies can be achieved manually by omitting certain directives. Bones and PGI Accelerator will always perform CPU-GPU and GPU-CPU copies, but they both allow the generated code to be edited to disable these intermediate copies.

6. CONCLUSION

In this work, we discussed four existing C-to-CUDA tools and introduced Bones, a new source-to-source compiler. The total of five tools were compared to each other in a head-to-head performance comparison for a set of 8 example image processing kernels. Bones is based on algorithmic skeletons and uses a modular and parameterisable algorithm classification which is introduced in this work. We conclude that Bones requires little modifications to the original source code, generates readable target code for further fine-tuning, and delivers superior performance in most cases compared to four existing tools for the example kernels.

We conclude that the other discussed tools (*hi*CUDA, SkePU, PGI Accelerator and Par4All) generate competitive code in a number of example cases. Especially PGI Accelerator and Par4All are promising, since the required input from the user is little or non-existent. Tools such as *hi*CUDA and SkePU do deliver competitive performance in

some cases, but require many code changes and/or additions, some of which are non-trivial. We furthermore believe that the practical applicability of tools such as *hi*CUDA, SkePU and PGI Accelerator is limited because further fine-tuning of the generated code is not feasible, either because the code is not available or difficult to read. With Bones, programmers are able to easily generate high performance CUDA or OpenCL code, while still being able to perform further fine-tuning if desired.

In this work, we have shown the potential of Bones for a limited number of examples. In future work, we aim to improve the practical usability of Bones by implementing more skeletons, fine-tuning existing skeletons, adding support for kernel fusion, and implementing a debug mode to check for correctness. Furthermore, in separate work, we focus on identifying algorithm classes from the source code using the polyhedral model, making it possible to fully automatically generate high performance target code using Bones.

7. ACKNOWLEDGEMENTS

We thank Sohan Walimbe for his help towards optimizing for *hi*CUDA and PGI Accelerator. He is a graduate student at Eindhoven University of Technology. We furthermore thank the authors of *hiCUDA* (David Han and Tarek Abdelrahman), SkePU (Usman Dastgeer) and Par4All (Mehdi Amini) for their help towards fine-tuning code for their tools and reviewing the corresponding sections in the paper.

8. REFERENCES

[1] M. Amini, F. Coelho, F. Irigoin, and R. Keryell. Static Compilation Analysis for Host-Accelerator Communication Optimization. In *LCPC '11: 24th International Workshop on Languages and Compilers for Parallel Computing*, 2011.

[2] S. Baghdadi, A. Größlinger, and A. Cohen. Putting Automatic Polyhedral Compilation for GPGPU to Work. In *CPC '10: 15th Workshop on Compilers for Parallel Computers*, 2010.

[3] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In *CC '10: 19th International Conference on Compiler Construction*. Springer Berlin, 2010.

[4] W. Caarls, P. Jonker, and H. Corporaal. Algorithmic Skeletons for Stream Programming in Embedded Heterogeneous Parallel Image Processing Applications. In *IPDPS '06: 20th International Parallel and Distributed Processing Symposium*. IEEE, 2006.

[5] D. K. G. Campbell. Towards the Classification of Algorithmic Skeletons. Technical Report YCS 276, University of York, 1996.

[6] B. Catanzaro, A. Fox, K. Keutzer, D. Patterson, B.-Y. Su, M. Snir, K. Olukotun, P. Hanrahan, and H. Chafi. Ubiquitous Parallel Computing from Berkeley, Illinois, and Stanford. *IEEE Micro*, 30:41–55, March 2010.

[7] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1991.

[8] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A Hybrid Multi-core Parallel Programming Environment. In *GPGPU-1: 1st Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[9] J. Enmyren and C. W. Kessler. SkePU: a Multi-backend Skeleton Programming Library for Multi-GPU Systems. In *HLPP '10: Fourth International Workshop on High-level Parallel Programming and Applications*. ACM, 2010.

[10] T. Han and T. Abdelrahman. hiCUDA: High-Level GPGPU Programming. *IEEE Transactions on Parallel and Distributed Systems*, 22:78 –90, Jan 2011.

[11] Y. He, Z. Ye, D. She, B. Mesman, and H. Corporaal. Feasibility Analysis of Ultra High Frame Rate Visual Servoing on FPGA and SIMD Processor. In *ACIVS '11: Advanced Concepts for Intelligent Vision Systems*. Springer Berlin, 2011.

[12] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31:7–17, September 2011.

[13] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *SC '10: Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2010.

[14] G. Noaje, C. Jaillet, and M. a. Krajecki. Source-to-Source Code Translator: OpenMP C to CUDA. In *HPCC: 13th International Conference on High Performance Computing and Communications*. IEEE, 2011.

[15] C. Nugteren and H. Corporaal. A Modular and Parameterisable Classification of Algorithms. Technical Report No. ESR-2011-02, Eindhoven University of Technology, 2011.

[16] C. Nugteren, H. Corporaal, and B. Mesman. Skeleton-based Automatic Parallelization of Image Processing Algorithms for GPUs. In *SAMOS XI: International Conference on Embedded Computer Systems*. IEEE, 2011.

[17] C. Nugteren, G.-J. van den Braak, H. Corporaal, and B. Mesman. High Performance Predictable Histogramming on GPUs: Exploring and Evaluating Algorithm Trade-offs. In *GPGPU-4: 4th Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2011.

[18] S. Sato and H. Iwasaki. A Skeletal Parallel Framework with Fusion Optimizer for GPGPU Programming. In *APLAS '09: 7th Asian Symposium on Programming Languages and Systems*. Springer Berlin, 2009.

[19] F. Seinstra, D. Koelma, and J. Geusebroek. A Software Architecture for User Transparent Parallel Image Processing. *Parallel Computing*, 28:967 – 993, 2002.

[20] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *IPDPSW '11: International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. IEEE, 2011.

[21] S.-Z. Ueng, M. Lathara, S. Baghsorkhi, and W.-m. Hwu. CUDA-Lite: Reducing GPU Programming Complexity. In *LCPC '08: 21th International Workshop on Languages and Compilers for Parallel Computing*. Springer Berlin, 2008.

[22] D. Unat, X. Cai, and S. B. Baden. Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C. In *ICS '11: International Conference on Supercomputing*, 2011.

[23] M. Wolfe. Implementing the PGI Accelerator model. In *GPGPU-3: 3rd Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2010.