Skeleton-based Automatic Parallelization of Image Processing Algorithms for GPUs

Cedric Nugteren, Henk Corporaal, Bart Mesman
Eindhoven University of Technology, The Netherlands
{c.nugteren, h.corporaal, b.mesman}@tue.nl

***Abstract*—Graphics Processing Units (GPUs) are becoming increasingly important in high performance computing. To maintain high quality solutions, programmers have to efficiently parallelize and map their algorithms. This task is far from trivial, leading to the necessity to automate this process.**

In this paper, we present a technique to automatically parallelize and map sequential code on a GPU, without the need for code-annotations. This technique is based on skeletonization and is targeted at image processing algorithms. Skeletonization separates the structure of a parallel computation from the algorithm's functionality, enabling efficient implementations without requiring architecture knowledge from the programmer. We define a number of skeleton classes, each enabling GPU specific parallelization techniques and optimizations, including automatic thread creation, on-chip memory usage and memory coalescing.

Recently, similar skeletonization techniques have been applied to GPUs. Our work uses domain specific skeletons and a finer-grained classification of algorithms. Comparing skeleton-based parallelization to existing GPU code generators in general, we potentially achieve a higher hardware efficiency by enabling algorithm restructuring through skeletons.

In a set of benchmarks, we show that the presented skeleton-based approach generates highly optimized code, achieving high data throughput. Additionally, we show that the automatically generated code performs close or equal to manually mapped and optimized code. We conclude that skeleton-based parallelization for GPUs is promising, but we do believe that future research must focus on the identification of a finer-grained and complete classification.

I. Introduction

Multi and many-core architectures are becoming a hot topic in the fields of computer architecture and high-performance computing. Processor design is no longer driven by high clock frequencies. Instead, more and more programmable cores are becoming available, even for low-power and embedded processors. At the same time, Graphics Processing Units (GPUs) are becoming increasingly programmable. While their graphics pipeline was completely fixed a few years ago, now, we are able to program all main processing elements using C-like languages.

Multi-core and many-core architectures are emerging and becoming a commodity. Many experts believe heterogeneous processor platforms including both GPUs and CPUs will be the future trend [15]. Already now, signs of this trend are present when we look at programming languages such as NVIDIA's CUDA or OpenCL.

Languages such as CUDA are specifically designed for general purpose GPU programming. Even so, the mapping process of most applications remains non-trivial. Furthermore, there is only a tiny fraction of programmers able to achieve an optimal hardware usage for their applications. Achieving this is non-trivial, having to deal with mappings of distributed memories, caches, and register files. Additionally, the programmer is exposed to parallel computing problems such as data-dependencies, race-conditions, synchronization barriers and atomic operations. As an example, we observe that a highly optimized GPU implementation of a reduction algorithm is about 30x faster than a naive implementation [10].

In the future, many programmers need to have access to the GPU's computing power, while nowadays only a fraction of programmers is able to benefit from these resources. Therefore, in order to use heterogeneous platforms efficiently, we have to change GPU programming. One way to achieve this is to automatically generate high performance code, preferably without requiring input (such as annotations) from the programmer.

In this paper, we focus on image processing algorithms, tackling the problem of how to generate high performance GPU code from sequential CPU code. We use skeleton-based parallelization, enabling the generation of GPU code based on sequential CPU code, requiring no programmer's knowledge on the architecture nor on the programming language. The contributions of this paper are as follows:

- Image processing algorithms are classified into a number of fine-grained *skeleton* classes. Classification is based on a study of the OpenCV library.
- For each class, a *skeleton implementation* is presented. To use the GPU's hardware efficiently, we generate optimized code. This includes among others high off-chip memory throughput and the efficient use of on-chip shared memory.
- We evaluate the limitations of the presented technique and discuss the future of skeleton-based parallelization for GPUs.

This paper is organized as follows. First, related work is discussed in section II. A motivation for the used techniques is found in section III. Section IV introduces GPU programming and image processing. Section V presents the code generation techniques. Next, section VI introduces and discusses the skeleton implementations. Section VII elaborates on the efficient use of GPU hardware, presents the results of skeleton-based parallelization, and discusses future work. Section VIII concludes the paper.

II. Related work

Keutzer et al. distinguish three solutions to the parallel programming problem [11]. The first two, automatic parallelization and new languages, have been applied to GPUs, but, according to the same authors, will not solve the problem. This section briefly introduces related work for these two solutions, but also for a third solution: skeleton-based parallelization.

First, we discuss the subject of automatically generating GPU code. For example, the source-to-source compilers CUDAlite [21] and hiCUDA [9] generate GPU code, but both require annotations in the original code. Similar techniques are used in the PGI accelerator [22], which uses OpenMP-style pragmas. The commonality between these automatic parallelization tools, is found in the fact that they apply transformations to the code where possible, but cannot, in contrast to this work, change the complete structure of an algorithm (e.g. the PGI accelerator does not support reduction [22]).

Numerous new programming languages for GPUs are introduced. For example, both Lee et al. [12] and Svensson et al. [19] present a functional language. Others, such as Cornwall et al. introduce a new framework [6]. These solutions all introduce a learning curve, reducing the chance for adoption by programmers.

Cole [5] introduced skeleton-based parallelization. Later, it has been applied for image processing algorithms. Examples target XeTaL and Imagine processors [4] [3] and FPGAs [2]. A related technique, based on patterns, has been presented for SIMD processors in general by Manniesing et al. [13]. Recently, skeleton-based parallelization has been applied to GPUs for a limited amount of skeletons in [18] and [7]. Our work uses domain specific skeletons and a finer-grained classification.

III. Motivation

Skeleton-based parallelization revolves around a set of parameterizeable *skeleton implementations*. At the same time, algorithms are categorized under *skeleton classes*. Corresponding to each class is a skeleton implementation and a set of *rewrite rules*, providing the means to automatically parallelize and map algorithms onto a target platform.

To justify the use of skeletonization, we compare the technique to a number of techniques with similar goals:

- Compared to a library-based approach, *skeletonization* can achieve higher flexibility (a library supports a finite amount of parameters), and can be applied to a wider range of algorithms (a library will not easily cover all desired algorithms).
- Skeletonization can achieve a higher performance compared to auto-parallelization in general. Since more knowledge of the algorithm is available, the algorithmic structure can be adjusted and optimizations can be performed, leading to the generation of highly efficient code.
- Auto-parallelization techniques based on annotations, such as pragma's, require input from the application programmer. Skeleton-based parallelization does not require programmers to annotate their code, they must only select a class based on high-level characteristics of the algorithm (e.g. data access patterns).

Skeleton-based parallelization creates a flexible code generation environment: for prior unsupported algorithms, new skeletons can be defined. Additionally, skeletons for different architectures can be included, serving as a base for automatic code generation targeting a range of heterogeneous multiprocessor platforms. This work presents a new code generation environment rather than extending existing work, providing a high degree of freedom to use non-traditional classification techniques and custom code generation techniques.

Although skeleton-based parallelization seems promising so far, there are a few caveats. First of all, porting an algorithm onto a different architecture often requires a change of the algorithm itself. One algorithm might be the best choice for a certain architecture, another algorithm (with the same functionality), might be more suitable for another architecture. A second caveat concerns the classification of algorithms. When just a few classes are defined, skeleton implementations might become too general to achieve high performance. However, when defining too many classes, the effort of implementing all skeletons converges towards implementing all algorithms.

IV. Background

For a full understanding of the problem and the solution, we discuss several background topics. In this section, we introduce a number of GPU programming concepts, followed by background on image processing. For an overview of the GPU architecture, we refer to [8] and figure 1.

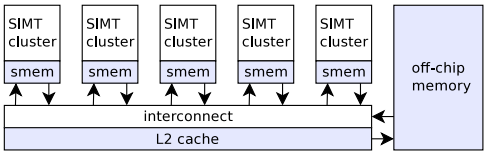

Fig. 1. Schematic view of a GPU architecture, consisting of a number of Single Instruction, Multiple Thread (SIMT) processor clusters. Each cluster has access to an on-chip shared memory (*smem*), a L2 cache and a large off-chip DDR memory.

A. GPU programming

We use the CUDA programming language as a target GPU language, since it supports many C++ features, has a large user-base, and provides well maintained compilers and simulators. Important CUDA GPU programming concepts are summarized below:

- The portions of code within a program that are executed on a GPU are called **kernels**. The programmer specifies a number of **threads**, which all execute the kernel code on different data. These threads are divided into groups of **threadblocks** by the programmer.
- Threads are scheduled by the hardware as **warps** in an SIMD-manner, typically consisting of 32 threads.

- It is important to enable a large number of simultaneously **active threads** on a throughput oriented architecture such as the GPU [8]. The number of active threads can be limited by among others register usage, threadblock size, and shared memory usage [14].
- The hardware is able to speed-up off-chip memory accesses significantly by combining multiple accesses. To enable this so-called **memory coalescing**, the kernel code has to meet certain requirements.
- To exploit data locality and to communicate within threadblocks, an on-chip **shared memory** is available.

B. Image processing

Image processing applications can be considered as structures of lower-level image processing algorithms, each processing one or more images and generating either one or more images or a set of characteristics. Image processing is closely related to computer vision, in which the goal of an application is to extract features of an input image. Most of these image processing algorithms require a large number of independent computations, making them suitable for a many-core architecture such as a GPU.

Libraries such as GPUCV [1], NVIDIA Performance Primitives (NPP) and CUVIlib [20] provide GPU implementations for a range of algorithms.

V. Code generation

Code generation is performed by first selecting a skeleton implementation from a pool. This is based on the class of algorithm, which is determined manually. Next, the skeleton is instantiated and passed through a selection of rewrite rules. This is in contrast to existing methods, which use C++ templates [7]. When generating code instead of using templates, code can be further fine-tuned where necessary after code generation.

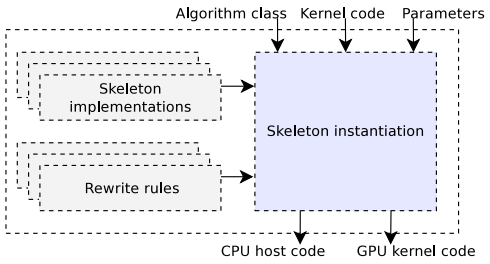

Fig. 2. An overview of skeleton-based code generation targeting GPUs.

An overview of the process is shown in figure 2. To illustrate it, we take the example of image convolution. Convolution belongs to a certain *algorithm class*, but individual algorithms may have different functionality (*kernel code* and *parameters*). Algorithms within this class correspond to a similar hardware efficient mapping, being a parameterizeable *skeleton implementation*. Using a set of *rewrite rules*, the skeleton implementation is instantiated, generating *CPU host code* (to start the kernel) and the *GPU kernel code* itself.

This section introduces the skeleton classes we defined for image processing applications, and presents details on the rewrite rules and their use.

A. Skeleton classes

We categorize low-level image processing algorithms into classes depending on their memory access patterns and data re-use characteristics. A significant number of higher level image processing algorithms and complete image processing applications can be decomposed into a sequence of these lower level algorithms [3]. In this paper we consider four classes, which are illustrated in figures 3(a), 3(b), 3(c) and 3(d). We give a short description of their characteristics:

- Pixel to pixel (**P2P**) algorithms generate one output pixel for each input pixel. Each output pixel must be generated by one unique input pixel, which typically has the same coordinates.
- For the neighbourhood to pixel (**N2P**) class, one output is generated for each input pixel. In contrast to P2P, the output value must be based on input pixels surrounding the coordinate of the output pixel. The size of this surrounding neighbourhood may vary.
- Reduction to scalar (**R2S**) algorithms generate a scalar value based on the complete set of input pixels.
- The reduction to vector (**R2V**) class is similar to R2S, but outputs a vector instead of a scalar.

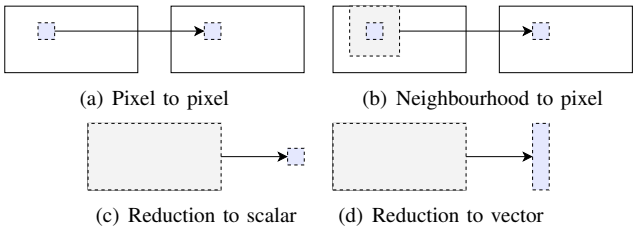

Fig. 3. Graphical overview of the categorization of image processing algorithms into classes.

To strengthen the case for the selection of these four classes, we analyzed algorithms from the OpenCV library. Algorithms from the library categories `image filtering`, `feature detection`, `motion & tracking`, `image transformation` and `histogram & matching` have been categorized as shown in table I. Algorithms denoted with `(C)` are considered compound algorithms, i.e. they can be split up in multiple lower-level algorithms. They therefore occur multiple times in table I. The algorithms that do not match the four presented classes can be categorized under classes defined by Caarls et al. [3]. The overview in table I shows that the presented set of classes includes the most common used image processing algorithms, but also leaves room for future implementations of other classes. We believe that this classification allows any algorithm designer to select the correct class.

TABLE I
OPENCV ALGORITHMS CLASSIFIED.

	Image Filtering	*Feature Detection*	*Motion & Tracking*	*Image Transformation, Histogram & Matching*
P2P	ImageSubtraction MorphologyEx (C)	-	Acc	Threshold CvtColor
N2P	Sobel Smooth IplConvKernel Laplace Dilate/Erode Filter2D MorphologyEx (C) PyrDown (C)	CornerMinEigenVal Canny FindCornerSubPix PreCornerDetect CornerHarris CornerEigenValsAndVecs GoodFeaturesToTrack (C) GetStarKeypoints (C) ExtractSURF (C)	CalcMotionGradient CalcOpticalFlowLK CalcOpticalFlowBM (C) CalcOpticalFlowPyrLK (C)	AdaptiveThreshold PyrSegmentation (C)
R2S	-	-	CalcOpticalFlowBM (C)	GetMinMaxHistValue
R2V	-	ExtractSURF (C) GetStarKeypoints (C)	CalcGlobalOrientation	CalcHist
Other	CopyMakeBorder PyrDown (C)	Houghlines SampleLine GoodFeaturesToTrack (C) GetStarKeypoints (C) ExtractSURF (C)	CalcOpticalFlowPyrLK (C)	Other kernels (25)

TABLE II
TWO REWRITE RULES AND EXAMPLE TRANSFORMATIONS.

Match	Replacement	Example transformation
`/#{name}\[w\s+\]\[h([^\]]+)\]/`	`/#{name}[p\\1]/`	`image_in[w][h-2]` → `image_in[p-2]`
`/smem_#{name}\[w\s+\]\[h([^\]]+)\]/`	`/smem_#{name}[smem_p\\1]/`	`in[w][h+3*str]` → `smem_in[smem_p+3*str]`

B. Rewrite rules

Corresponding to each skeleton class is a skeleton implementation and a set of rewrite rules. The latter are mainly used to generate the necessary correct code to instantiate the skeletons. Rewrite rules are based on pattern matching and implemented using regular expressions. These regular expressions match certain patterns (e.g. array indices) and replace them with other patterns (e.g. shared memory indices) under certain conditions (e.g. the skeleton is N2P).

Among others, rewrite rules serve the following purposes: (a) to enable the use of on-chip shared memory, (b) to enable the generation of boundary/special-case code, (c) to make use of the GPU's special function hardware, (d) to transform array dimensions of the streaming input and output into 1D arrays, and (e) to fine-tune the generated code (e.g. threadblock size). Since rewrite rules are mainly used to generate correct code instead of optimizing the code, they are independent of each other and we are free to apply them in any order.

We present two example rewrite rules (out of 18 total) in table II to illustrate their implementation, which is given by a *match* and a *replacement* regular expression in the Ruby language. In both examples, array variables are matched and replaced with updated array indices to match the skeleton implementations. While the first example translates array indices, the second performs off-chip to on-chip memory conversion, and is applied in certain cases, such as for the N2P class.

VI. SKELETON IMPLEMENTATIONS

In this section, we discuss the skeleton implementations. Additionally, an example R2S algorithm is presented, with listings of CUDA code.

A. Pixel to pixel

The P2P skeleton kernel is implemented using 4 lines of code. The most important characterisitcs of the P2P skeleton are:

- All operations are independent. Therefore, the amount of exploitable parallelism is at least[1] equal to the amount of pixels.
- Since the P2P class does not imply data re-use, on-chip shared memory is not used.
- Rewrite rules support a coalesced memory access pattern for both row-by-row and column-by-column accessing.

Although algorithms such as image mirroring, forward warping, and transpose also belong to the pixel to pixel skeleton, they do not read or write pixels sequentially. To still enable coalesced writing and reading, a second (closely related) skeleton implementation is introduced for this special case. It uses the on-chip shared memory to store intermediate results in order to access the off-chip memory coalesced.

[1]Parallelism can be higher depending on the algorithm

B. Neighbourhood to pixel

We discuss the N2P skeleton implementation in this section, which uses 18 lines of code. Despite the implementation not being as trivial as for the P2P skeleton, many documents describe efficient implementations (e.g. [17]), with convolution as a driving example. The most important characteristics of the N2P skeleton are:

- All operations are independent, and the amount of parallelism exploited is equal to the amount of pixels in the image (one thread per pixel).
- Since the N2P class implies local data re-use, we exploit on-chip shared memory within threadblocks. We schedule threadblocks as large as supported by the hardware to minimize the amount of borders between threadblocks. At the border, additional boundary pixels have to be loaded into on-chip shared memory. Rewrite rules translate memory addresses to make use of the on-chip shared memory.
- The skeleton implementation enables a coalesced memory access pattern when pre-loading data into the on-chip shared memory, enabling fast off-chip memory access. However, when loading boundary data, performance of off-chip memory can be limited due to non-sequential accesses.

Again, there is a caveat. As presented by Podlozhnyuk, special cases of N2P algorithms with separable filters can reduce non-coalesced boundary accesses and improve overall performance [17]. Although these algorithms can be implemented using the N2P skeleton, their performance might not be optimal. At this moment, such a special case skeleton is not implemented, however, programmers can fine-tune their algorithms after code generation.

C. Reduction to scalar

Reduction is a common type of operation, and a CUDA implementation has been documented by among others Harris in [10]. Our implementation (the kernel implementation contains 14 lines of code) follows the guidelines presented in this document, using a reduction tree in $\log_2(image\_size)$ steps. The main characteristics of the R2S skeleton implementation are:

- The amount of parallelism is equal to half the image size at the start, ending at the bottom of the tree with no parallelism at all. Therefore, we perform R2S using multiple kernels (dependent on the image size). Each kernel performs a number of steps (corresponding to strides) in the reduction tree.
- We use on-chip shared memory for intermediate storage and to ensure coalescing from off-chip memory. The skeleton is adjusted to avoid bank conflicts for the on-chip shared memory.

To further increase efficiency and reduce loop overhead, we unroll the loops as discussed by Harris [10]. As presented in the same document, we also reduce thread idling by loading two elements per thread and performing the first operation directly. This requires rewrite rules to generate specialized code that loads elements and computes the first operation directly.

An example of a 2-level 32-elements tree with 8 threads per threadblock is shown in figure 4. Each of the threads loads one pixel. Next, half of the threads perform the first operation (stride 0), followed by two (stride 1) and finally just one (stride 2). In the next level or kernel, we use the same pattern, but invoke only one threadblock.

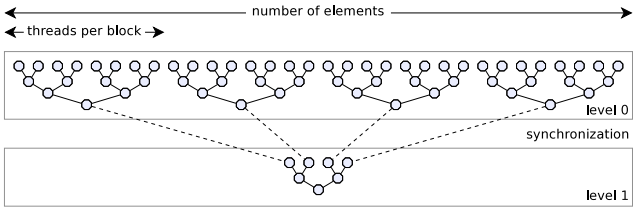

Fig. 4. Overview of a reduction tree. The tree is used to implement the reduction to scalar skeleton.

D. Reduction to vector

Another reduction operation, R2V, is less common. However, one example algorithm, histogramming, is described in detail by Podlozhnyuk in [16]. We implement our R2V skeleton according to this example. Similar to R2S, we can efficiently perform R2V in parallel using a reduction tree, using a slightly different structure. As seen in an example in figure 5, each threadblock now calculates a vector of *bins* (28 lines of code). After a global synchronization barrier, a second kernel is started, combining all the vectors into the final result (6 lines of code). To reduce instruction overhead, threads can process multiple data elements, the amount of which is determined by the architecture and image size, and implemented with a rewrite rule.

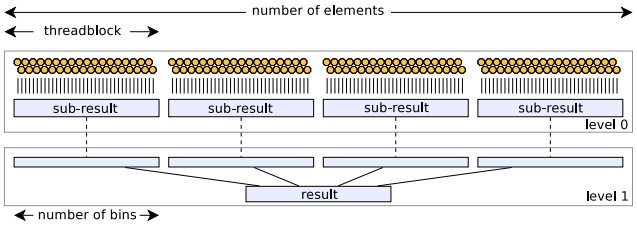

Fig. 5. To implement reduction to vector, a tree structured is used, dividing input elements over threadblocks.

The on-chip shared memory is fully exploited for R2S. Bins are stored in shared memory as they are updated frequently by possibly all threads in a threadblock. However, since the access patterns of threads to the bins stored in shared memory are dependent on the input data, *shared memory collisions* can occur, leading to incorrect results. Therefore, writing to the shared memory is performed in a do-while loop using a unique tag. Only when the data is successfully written, the do-while loop ends. Even then, collisions can still occur in between warps. To solve this, each warp is given its own

on-chip shared memory range and thus its own bins. This introduces an additional reduction step, in which all the bins of each warp need to be combined into one for each threadblock. This is illustrated in figure 6. Avoiding these on-chip memory collisions is discussed in more detail by Podlozhnyuk [16].

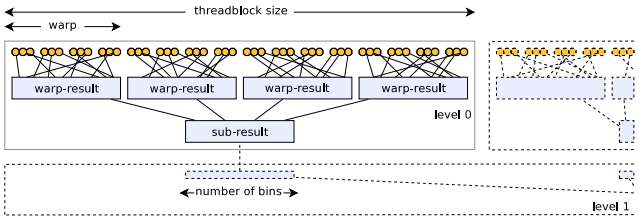

Fig. 6. Close-up of the R2V tree structure. Within a threadblock, a further division over warps is made.

By coalescing memory accesses, neighbouring pixels are processed together by the same warp. For images, these pixels typically have a high correlation. This can lead to a significant decrease in performance due to a higher chance of shared memory collisions compared with uncoalesced accesses. In contrast to Podlozhnyuk [16], we deliberately do not perform coalescing for R2V, benefiting from less restrictions on access patterns, and a lower correlation between pixels.

E. An illustrating example

This section presents example code listings. In these listings, function parameters are underlined and placeholders are highlighted. Variables to be replaced by rewrite rules are preceded by the '@' character. We simplified the presented code listings to ease readability (e.g. not showing unrolled code), and omitted the code necessary for kernel call(s) and threadblock sizing.

```
1 uint result = 0;
2 for (each pixel in image) {
3   if (result < pixel) {
4     result = pixel;
5   }
6 }
```

Listing 1. CPU pseudo code.

We take an example low-level image processing algorithm in order to illustrate the code generation techniques involved in skeleton-based parallelization. The example algorithm, finding the maximum pixel value in an image, is classified under the reduction to scalar class. The pseudo code for this algorithm is shown in listing 1. The kernel of the algorithm is highlighted in lines 3-5, and serves as input for code generation.

The code generator transforms lines 3-5 of listing 1 into listing 3, using the R2S skeleton implementation (shown in listing 2) and a set of rewrite rules. The generation of lines 18-20 is relatively straightforward, and is performed by: (1) off-chip to on-chip address transformation, and (2) storing intermediate results in the first element to create the tree structure. To generate lines 7-12, we use rewrite rules

```
1  // Identify the thread
2  uint p = blockIdx.x*blockDim.x*2+threadIdx.x;
3
4  // Load data and synchronize threads
5  __shared__ uint smem_in[1024];
6  uint smem_p = threadIdx.x;
7  @placeholder_0
8  __syncthreads();
9
10 // Start the calculation
11 for (uint s=blockDim.x/2; s>0; s=s>>1) {
12   if (smem_p < s) {
13     @placeholder_1
14   }
15   __syncthreads();
16 }
17
18 // Write result back to global memory
19 if (threadIdx.x == 0) {
20   result[blockIdx.x] = smem_in[0];
21 }
```

Listing 2. R2S implementation.

```
1  // Identify the thread
2  uint p = blockIdx.x*blockDim.x*2+threadIdx.x;
3
4  // Load data and synchronize threads
5  __shared__ uint smem_in[1024];
6  uint smem_p = threadIdx.x;
7  if (image_in[p] < image_in[p+blockDim.x]) {
8    smem_in[smem_p] = image_in[p+blockDim.x];
9  }
10 else {
11   smem_in[smem_p] = image_in[p];
12 }
13 __syncthreads();
14
15 // Start the calculation
16 for (uint s=blockDim.x/2; s>0; s=s>>1) {
17   if (smem_p < s) {
18     if (smem_in[smem_p] < smem_in[smem_p+s]) {
19       smem_in[smem_p] = smem_in[smem_p+s];
20     }
21   }
22   __syncthreads();
23 }
24
25 // Write result back to global memory
26 if (threadIdx.x == 0) {
27   result[blockIdx.x] = smem_in[0];
28 }
```

Listing 3. Generated code.

to: (1) transform off-chip to on-chip addresses, (2) execute the first computation, and (3) extend the computation with an else statement to ensure a load. Rewrite rules and the skeleton implementation ensure hardware efficiency according to NVIDIA guidelines [10], enabling sequential addressing, on-chip shared memory, coalesced loads, loop unrolling and limited bank conflicts.

VII. Evaluation

In this section, we discuss the obtained hardware efficiency of the presented technique, evaluate the usability of skeleton-based parallelization and discuss limitations and future work.

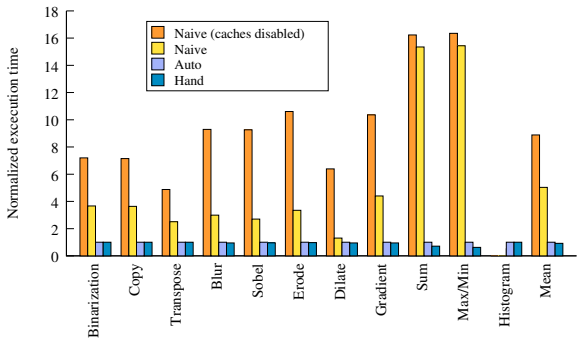

Fig. 7. Comparison of the execution time of generated GPU code against naive and hand optimized code.

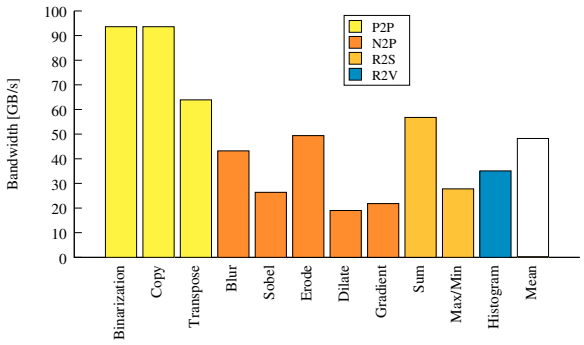

Fig. 8. Evaluation of hardware efficiency, in which off-chip memory bandwidth is used as a metric. The roofline is ±100GB/s.

The presented benchmarks use a Geforce GTX470 GPU with 448 CUDA cores in an Intel Core-i7 930 system. The GPU has a peak practical bandwidth of ±100GB/s. All results are averaged over 20 runs and are tested on random 8-bit data with dimensions of 2048x2048 pixels. Since no low-level image processing benchmark suite is available, we hand-picked benchmarks of common used low-level image processing algorithms and categorized them as follows:

- P2P Binarize, Copy, Transpose
- N2P Blur (3x3), Sobel (3x3), Erode (3x3), Dilate (5x5), Gradient (7x3)
- R2S Sum, Min, Max
- R2V Histogram

A. Performance evaluation

The first results show the normalized execution time[2] of each algorithm[3]. Figure 7 distinguishes a naive GPU implementation (with and without the cache), generated GPU code and hand optimized code. Since a cache is available on newer GPUs, on-chip shared memory usage is not strictly necessary to exploit data re-use. As an extreme example, the dilate algorithm that enables on-chip shared memory performs close to a naive implementation with hardware managed cache. In general, the benchmarks show a performance close or equal to hand optimized code for the generated code.

To measure the hardware efficiency, we benchmarked the performance in terms of bandwidth (see figure 8). The P2P algorithms map well on the GPU architecture, since a bandwidth close to the peak practical bandwidth is achieved. The P2P algorithm transpose shows a slight performance drop, as intermediate results have to be stored in the on-chip shared memory to guarantee memory coalescing. The five N2P algorithms show a further decrease in performance, as they load some data elements multiple times from the off-chip memory, which includes uncoalesced memory accesses on edges of threadblocks. As can be seen from the results of the dilate and gradient algorithms, hardware efficiency drops further when filter sizes increase. The reduction algorithms R2S and R2V still show a high bandwidth utilization, benefiting from their complex algorithm structures. When comparing the R2S algorithms sum and max/min, we argue that the latter is slower due to an if-statement in the original algorithm.

Lastly, we benchmarked image sizes ranging from 256K to 16M pixels. The algorithms closest to peak performance (P2P, R2S and R2V) show a significant performance degradation when using smaller images, since overhead is becoming significant (e.g. kernel launch overhead). For more compute intensive algorithms (N2P), this effect is less visible.

B. Usability

To evaluate the usability of the presented technique, we compare a naive CPU implementation against generated GPU code based on the same naive CPU code in table III. We do not intend to compare both architectures performance-wise with these benchmarks, we merely illustrate the potential benefit of accelerating these image processing algorithms using a GPU.

TABLE III
GPU ACCELERATION OF IMAGE PROCESSING.

algorithm	naive-CPU	auto-GPU	algorithm	naive-CPU	auto-GPU
Binarize	106 ms	0.34 ms	Dilate	445 ms	1.67 ms
Copy	99 ms	0.34 ms	Gradient	432 ms	1.45 ms
Transpose	88 ms	0.50 ms	Sum	37 ms	0.28 ms
Blur	208 ms	0.74 ms	Max	41 ms	0.57 ms
Sobel	230 ms	1.21 ms	Min	41 ms	0.56 ms
Erode	151 ms	0.65 ms	Histogram	213 ms	0.47 ms

The benchmark set consists of algorithms with up to 10 lines of code and limited functionality (e.g. no function calls, library calls, classes). The presented technique is intended to work for low-level algorithms, consisting of small and simple inner-loop code. Compound kernels or full applications can be handled by the programmer by splitting them into multiple low-level kernels.

An issue that addresses usability is the availability of suitable classes. A problem can arise when: (1) an algorithm cannot be classified under any available skeleton type, or (2) a skeleton is too generic to generate hardware efficient code. The

[2]Normalized against the performance of generated code
[3]The histogram algorithm does not map trivially on the GPU [16]

first issue is solved by implementing a new skeleton. This is only useful when multiple algorithms can be categorized under the same class. The latter issue (generic skeletons) is addressed by the possibility of fine-tuning the generated CUDA code where needed.

C. Discussion and future work

Many forms of automatic parallelization, such as this work, are potentially promising. Although skeleton-based parallelization has clear benefits over other techniques (pragma-based techniques require annotations, new languages introduce a learning curve) [11], it is not been used widely in practice. The main reason for this is the limited applicability of the technique. While the presented benchmarks show good results for these algorithms, they might not for others. For example, algorithms might belong to different classes not implemented or defined yet, or they might be a special case of an implemented class requiring further fine-tuning. In the latter case, the presented technique in this paper has an advantage over template-based skeletonization (as presented in [7]), in which no fine-tuning is possible after the instantiation of the skeleton.

To reduce the limited applicability, a good classification is paramount. Existing skeleton-based parallelization for GPUs uses classes such as map, reduce and map-reduce [7] [18]. To achieve a finer-grained classification, this work uses a domain specific classification, yielding high hardware efficiency. Still, the presented set already identifies a number of caveats (non-sequential P2P, separable N2P), while supporting only a set of image processing algorithms. In order to find practical applicability for skeleton-based parallelization, we believe that future research must focus on the identification of a fine-grained, modular, and complete classification.

VIII. Conclusions

In this work, we introduced a number of algorithmic skeleton implementations for many-core GPU architectures, categorized as classes. Together with a set of rewrite rules, the algorithmic skeletons enable code generation, both parallelizing sequential code and mapping the resulting code onto a GPU. In contrast to existing GPU code generators, a skeleton-based technique enables the restructuring of algorithms, enabling a hardware efficient mapping.

We compared the performance of 12 algorithms against the peak performance of a GPU in terms of off-chip memory bandwidth. In another measurement, we compared the algorithms with naive and hand optimized code. The generated code runs 8 times faster compared to naive code and performs at 93% of the performance of hand optimized code on average.

Although the presented technique is usable for a set of domain specific algorithms, we believe that skeleton-based parallelization techniques for GPUs will not be widely used until a complete set of algorithms can be supported through a fine-grained classification. Although the domain of applicability of this work is still limited, we do set another step towards low effort, high performance GPU programming.

IX. Acknowledgements

We thank Zhenyu Ye for his classification of OpenCV algorithms into skeleton types. He is with the Eindhoven University of Technology.

References

[1] Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan. GpuCV: an opensource GPU-accelerated framework for image processing and computer vision. In *MM '08: Proceeding of the 16th ACM international conference on Multimedia*, pages 1089–1092. ACM, 2008.

[2] K. Benkrid, D. Crookes, and A. Benkrid. Towards a general framework for FPGA based image processing using hardware skeletons. *Parallel Computing*, 28(7-8):1141 – 1154, 2002.

[3] W. Caarls. *Automated Design of Application-Specific Smart Camera Architectures*. PhD thesis, Eindhoven University of Technology, 2008.

[4] W. Caarls, P. Jonker, and H. Corporaal. Algorithmic skeletons for stream programming in embedded heterogeneous parallel image processing applications. In *Parallel and Distributed Processing Symposium. IPDPS 2006. 20th International*, page 9 pp., 2006.

[5] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991.

[6] J. L. Cornwall, L. Howes, P. H. Kelly, P. Parsonage, and B. Nicoletti. High-performance SIMT code generation in an active visual effects library. In *CF '09: Proceedings of the 6th ACM conference on Computing frontiers*, pages 175–184. ACM, 2009.

[7] J. Enmyren and C. W. Kessler. SkePU: a multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[8] M. Garland and D. B. Kirk. Understanding throughput-oriented architectures. *Communications of the ACM*, 53:58–66, November 2010.

[9] T. D. Han and T. S. Abdelrahman. 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*, pages 52–61. ACM, 2009.

[10] M. Harris. Optimizing parallel reduction in CUDA. Technical report, NVIDIA, 2008. NVIDIA.

[11] K. Keutzer and T. Mattson. A Design Pattern Language for Engineering (Parallel) Software. In *Intel Technology Journal*, 2010.

[12] S. Lee, V. Grover, M. Chakravarty, and G. Keller. GPU Kernels as Data-Parallel Array Computations in Haskell. In *EPHAM 09': Exploiting Parallelism using GPUs and other Hardware-Assisted Methods*, 2009.

[13] R. Manniesing, I. Karkowski, and H. Corporaal. Automatic SIMD Parallelization of Embedded Applications Based on Pattern Recognition. In A. Bode, T. Ludwig, W. Karl, and R. Wismuller, editors, *Euro-Par 2000 Parallel Processing*, pages 349–356, 2000.

[14] C. Nugteren, B. Mesman, and H. Corporaal. Analyzing CUDA's Compiler through the Visualization of Decoded GPU Binaries. In *ODES-8: Proceedings of the 8th Workshop on Optimizations for DSP and Embedded Systems at CGO '10*, 2010.

[15] D. Patterson. The Top 10 Innovations in the New Fermi Architecture, and the Top 3 Next Challenges. NVIDIA Whitepaper, 2009.

[16] V. Podlozhnyuk. Histogram calculation in CUDA. Technical report, NVIDIA, 2007.

[17] V. Podlozhnyuk. Image Convolution with CUDA. Technical report, NVIDIA, 2007.

[18] S. Sato and H. Iwasaki. A Skeletal Parallel Framework with Fusion Optimizer for GPGPU Programming. In Z. Hu, editor, *Programming Languages and Systems*, volume 5904 of *Lecture Notes in Computer Science*, pages 79–94. Springer Berlin Heidelberg, 2009.

[19] J. Svensson, K. Claessen, and M. Sheeran. GPGPU kernel implementation and refinement using Obsidian. *Procedia Computer Science*, 1(1):2059 – 2068, 2010. ICCS 2010.

[20] TunaCode. CUVIlib. http://www.cuvilib.com.

[21] S.-Z. Ueng, M. Lathara, S. Baghsorkhi, and W.-m. Hwu. CUDA-Lite: Reducing GPU Programming Complexity. In *Languages and Compilers for Parallel Computing*, volume 5335 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin, 2008.

[22] M. Wolfe. Implementing the PGI Accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 43–50, USA, 2010. ACM.