MetaFork: A Compilation Framework for Concurrency Models Targeting Hardware Accelerators and Its Application to the Generation of Parametric CUDA Kernels

Changbo Chen
CIGIT, Chinese Academy of Sciences
changbo.chen@hotmail.com

Xiaohui Chen
University of Western Ontario & IBM Canada Laboratory, CAS Research
xchen422@uwo.ca

Abdoul-Kader Keita
IBM Canada Laboratory
akkeita@ca.ibm.com

Marc Moreno Maza
CIGIT, Chinese Academy of Sciences & University of Western Ontario & IBM Canada Laboratory, CAS Research
moreno@csd.uwo.ca

Ning Xie
University of Western Ontario
nxie6@csd.uwo.ca

ABSTRACT

In this paper, we present the accelerator model of MetaFork together with the software framework that allows automatic generation of CUDA code from annotated MetaFork programs. One of the key features of this CUDA code generator is that it supports the generation of CUDA kernel code where program parameters (like number of threads per block) and machine parameters (like shared memory size) are allowed. These parameters need not to be known at code-generation-time: machine parameters and program parameters can be respectively determined and optimized when the generated code is installed on the target machine.

This generation of parametric CUDA kernels requires from the MetaFork framework to deal with non-linear polynomial expressions during the dependence analysis and tiling phase of the MetaFork code. To achieve these algebraic calculations, we take advantage of quantifier elimination and its implementation in the `RegularChains` in Maple. Various illustrative examples are provided together with performance evaluation.

1. INTRODUCTION

In the past decade, the introduction of low-level heterogeneous programming models, in particular CUDA, has brought supercomputing to the level of the desktop computer. However, these models bring notable challenges, even to expert programmers. Indeed, fully exploiting the power of hardware accelerators with CUDA-like code often requires significant code optimization effort. While this development can indeed yield high performance, it is desirable for some programmers to avoid the explicit management of device initialization and data transfer between memory levels. To this end, high-level models for accelerator programming have become an important research direction. With these models, programmers only need to annotate their C/C++ code to indicate which code portion is to be executed on the device and how data maps between host and device.

As of today, OpenMP and OpenACC are among the most developed accelerator programming models. Both OpenMP and OpenACC are built on a host-centric execution model. The execution of the program starts on the host and may offload target regions to the device for execution. The device may have a separated memory space or may share memory with the host, so that memory coherence is not guaranteed and must be handled by the programmer. In OpenMP and OpenACC, the division of the work between thread blocks within a grid and, between threads within a thread block can be expressed in a loose manner, or even ignored. This implies that code optimization techniques may be applied in order to derive efficient CUDA-like code.

MetaFork is a high-level programming language extending C/C++, which combines several models of concurrency including fork-join and pipelining parallelisms. MetaFork is also a compilation framework which aims at facilitating the design and implementation of concurrent programs through three key features:

(1) Perform automatic code translation between concurrency platforms targeting both multi-core and many-core GPU architectures.
(2) Provide a high-level language for expressing concurrency as in the fork-join model, the SIMD (Single Instruction Multiple Data) paradigm and the pipelining parallelism.
(3) Generate parallel code from serial code with an emphasis on code depending on machine or program parameters (e.g. cache size, number of processors, number of threads per thread block).

CASCON'15 November 2-4, 2015, Toronto, Canada

As of today, the publicly available and latest release of MetaFork, see `www.metafork.org`, offers the second feature stated above, as well as the multi-core portion of the first one. To be more specific, MetaFork is a meta-language for concurrency platforms, based on the fork-join model and pipelining parallelism, which targets multi-core architectures. This meta-language forms a bridge between actual multithreaded programming languages and we use it to perform automatic code translation between those languages, which, currently consist of CilkPlus and OpenMP, see [9].

In this paper, we present the accelerator model of MetaFork together with the software framework that allows automatic generation of CUDA code from annotated MetaFork programs. One of the key properties of this CUDA code generator is that it supports the generation of CUDA kernel code where program parameters (like number of threads per block) and machine parameters (like shared memory size) are allowed. These parameters need not to be known at code-generation-time: machine parameters and program parameters can be respectively determined and optimized when the generated code is installed on the target machine. Therefore, MetaFork now offers a preliminary implementation of the third feature stated above, as well as the many-core GPU portion of the first one.

The need for CUDA programs (more precisely, kernels) depending on program parameters and machine parameters is argued in Section 2.

In Section 3, following the authors of [16], we observe that generating *parametric* CUDA kernels require the manipulation of systems of non-linear polynomial equations and the use of techniques like quantifier elimination (QE). To this end, we take advantage of the `RegularChains` library of Maple [8] and its QuantifierElimination command which has been designed to efficiently support the non-linear polynomial systems coming from automatic parallelization.

Section 4 is an overview of the concurrency models offered by MetaFork. In particular, the MetaFork language constructs for generating SIMD code (in languages targeting many-core GPU architectures, like CUDA) are discussed.

Section 5 reports on a preliminary implementation of the MetaFork generator of *parametric* CUDA kernels from input MetaFork programs. In addition to the `RegularChains` library, we take advantage of PPCG, the polyhedral parallel code generation for CUDA [33] that we have adapted so as to produce parametric CUDA kernels.

Finally, Section 6 gathers experimental data demonstrating the performance of our generated parametric CUDA code. Not only these results show that the generation of parametric CUDA kernels helps optimizing code independently of the values of the machine parameters of the targeted hardware, but also these results show that automatic generation of parametric CUDA kernels may discover better values for the program parameters than those computed by a tool generating non-parametric CUDA kernels.

2. OPTIMIZING CUDA KERNELS DEPENDING ON PROGRAM PARAMETERS

Estimating the amount of computing resource (time, space, energy, etc.) that a parallel program, written in a high-level language, required to run on a specific hardware is a well-known challenge. A first difficulty is to define models of computation retaining the computer hardware characteristics that have a dominant impact on program performance. That is, in addition to specify the appropriate complexity measures, those models must consider the relevant parameters characterizing the abstract machine executing the algorithms to be analyzed. A second difficulty is, for a given model of computation, to combine its complexity measures so as to determine the "best" algorithm among different algorithms solving a given problem. Models of computation which offer those estimates necessarily rely on simplification assumptions. Nevertheless, such estimates can deliver useful predictions for programs satisfying appropriate properties.

An instance of such model of computation is the fork-join concurrency model [7] where two complexity measures, the work T_1 and the span T_∞, and one machine parameter, the number P of processors, are combined into a running time estimate by means of the Graham-Brent theorem [7, 14]. A refinement of this theorem supports the implementation on *multi-core architectures* of the parallel performance analyzer `Cilkview` [19]. In this context, the running time T_{P} is bounded in expectation by $T_1/\mathsf{P} + 2\delta\widehat{T_\infty}$, where δ is a constant (called the *span coefficient*) and $\widehat{T_\infty}$ is the burdened span, which captures parallelism overheads due to scheduling and synchronization.

Turning our attention to many-core GPUs, the fork-join concurrency model becomes inappropriate to simulate the SIMD execution model of CUDA kernels. The PRAM (Parallel Random-Access Machine) model [31, 12] and its extensions, like that reported in [1] (which integrates communication delay into the computation time) and that presented in [13] (which integrates memory contention) captures the SIMD execution model. However, a PRAM abstract machine consists of an unbounded collection of RAM processors accessing a global memory, whereas a many-core GPU holds a *collection* of streaming *multiprocessors* (SMs) communicating through a global memory. Thus, a many-core GPU implements a combination of two levels of concurrency.

Recent works have further extended the RAM model in order to better support the analysis of algorithms targeting implementation on many-core GPUs. Ma, Agrawal and Chamberlain [24] introduce the TMM (Threaded Many-core Memory) model which retains many important characteristics of GPU-type architectures as machine parameters, such as memory access width and hardware limit on number of threads per core.

In [18], we propose a many-core machine (MCM) model for multithreaded computation combining the fork-join and SIMD parallelisms; a driving motivation in this work is to estimate parallelism overheads (data communication and synchronization costs) of GPU programs. In practice, the MCM model determines a trade-off among *work*, *span* and *parallelism overhead* by checking the estimated overall running time so as to (1) either tune a program parameter or, (2) compare different algorithms independently of the hardware details. We illustrate the use of the MCM model with a very classical example: the computation of Fast Fourier Transforms (FFTs). Our goal is to compare the running times of two of the most commonly used FFT algorithms: that of Cooley & Tukey [11] and that of Stockham [30].

Let f be a vector over a field K of coefficients, say the complex numbers. Assume that f has size n where n is a power of 2. Let U be the time (expressed in clock cycles) to transfer one machine word between the global memory and the private memory of any SM, that is, $U > 0$. Let

Z be the size (expressed in machine words) of the private memory of any SM, which sets up an upper bound on several program parameters. Let $\ell \geq 2$ be a positive integer. For Z large enough, both Cooley & Tukey algorithm and Stockham algorithm can be implemented

- by calling $\log_2(n)$ times a kernel using $\Theta(\frac{n}{\ell})$ SMs[1] each of those SMs executing a thread-block with ℓ threads,
- with a respective *work*[2] of $\mathrm{W}_{ct} = n\,(34\,\log_2(n)\,\log_2(\ell) + 47\,\log_2(n) + 333 - 136\log_2(\ell))$ and $\mathrm{W}_{sh} = 43\,n\,\log_2(n) + \frac{n}{4\,\ell} + 12\,\ell + 1 - 30\,n$,
- with a respective *span*[3] of $\mathrm{S}_{ct} = 34\,\log_2(n)\,\log_2(\ell) + 47\,\log_2(n) + 2223 - 136\log_2(\ell)$ and $\mathrm{S}_{sh} = 43\,\log_2(n) + 16\,\log_2(\ell) + 3$,
- with a respective *overhead*[4] of $\mathrm{O}_{ct} = 2\,n\,U\,(\frac{4\,\log_2(n)}{\ell} + \log_2(\ell) - \frac{\log_2(\ell)+15}{\ell})$ and $\mathrm{O}_{sh} = \frac{5\,n\,U\,\log_2(n)}{\ell} + \frac{5\,n\,U}{4\,\ell}$.

See [18] for details on the above estimates. From those, one observes that the overhead of Cooley & Tukey algorithm has an extraneous term in $O(nU\log_2(\ell))$ which is due to higher amount of non-coalesced accesses. In addition, when n escapes to infinity (while ℓ remains bounded over on a given machine since we have $\ell \in O(Z)$) the work and span of the algorithm of Cooley & Tukey are increased by a $\Theta(\log_2(\ell))$ factor w.r.t their counterparts in Stockham algorithm.

These theoretical observations suggest that, as ℓ increases, Stockham algorithm performs better than the one of Cooley & Tukey. This has been verified experimentally[5] by the authors of [18] as well as by others, see [25] and the papers cited therein. On the other hand, it was also observed experimentally that for ℓ small, Cooley & Tukey algorithm is competitive with that of Stockham. Overall, this suggests that generating kernel code, for both algorithms, where ℓ is an input parameter, is a desirable goal. With such parametric codes, one can choose at run-time the most appropriate FFT algorithm, once ℓ has been chosen.

The MCM model retains many of the characteristics of modern GPU architectures and programming models, like CUDA [26, 27] and OpenCL [32]. However, in order to support algorithm analysis with an emphasis on parallelism overheads, the MCM abstract machines admit a few simplifications and limitations with respect to actual many-core devices.

To go further in our discussion of CUDA kernel performance, let us consider now the programming model of CUDA itself and its differences w.r.t. the MCM model. In CUDA, instructions are issued per *warp*; a warp consists of a fixed number S_{warp} of threads. Typically S_{warp} is 32 and, thus, executing a thread-block on an SM means executing several warps in turn. If an operand of an executing instruction is not ready, then the corresponding warp stalls and context switch happens between warps running on the same SM.

Registers and shared memory are allocated for a thread-block as long as that thread-block is active. Once a thread-block is active it will stay active until all threads in that thread-block have completed. Context switching is very fast because registers and shared memory do not need to be saved and restored. The intention is to hide the latency (of data transfer between the global memory and the private memory of an SM) by having more memory transactions in fly. There is, of course, a hardware limitation to this, characterized by (at least) two numbers:

- the maximum number of active warps per SM, denoted here by M_{warp}; A typical value for M_{warp} is 48 on a Fermi NVIDIA GPU card, leading to a maximum number of $32 \times 48 = 1536$ active threads per SM.
- the maximum number of active thread blocks per SM, denoted here by M_{block}; A typical value for M_{block} is 8 on a Fermi NVIDIA GPU card.

One can now define a popular performance counter of CUDA kernels, the *occupancy* of an SM: it is given by $A_{\mathrm{warp}}/M_{\mathrm{warp}}$, where A_{warp} is the number of active warps on that SM. Since resources (registers, shared memory, thread slots) are allocated for an entire thread-block (as long as that block is active) there are three potential limitations to occupancy: register usage, shared memory usage and thread-block size. As in our discussion of the MCM model, we denote the thread-block size by ℓ. Two observations regarding the possible values of ℓ:

- The total number of active threads is bounded over by $M_{\mathrm{block}}\,\ell$, hence a small value for ℓ may limit occupancy.
- A larger value for ℓ will reduce the amount of registers and shared memory words available per thread; this will limit data reuse within a thread-block and, thus, will potentially increase the amount of data transfer between global memory and the private memory of an SM.

Overall, this suggests again that generating kernel code, where ℓ, and other program parameters are input arguments, is a desirable goal. With such parametric code at hand, one can optimize at run-time the values of those program parameters (like ℓ) once the machine parameters (like S_{warp}, M_{warp}, M_{block}, Z (private memory size) and the size of the register file) are known.

3. AUTOMATIC GENERATION OF PARAMETRIC CUDA KERNELS

The general purpose of automatic parallelization is to convert sequential computer programs into multithreaded or vectorized code. Following the discussion of Section 2, we are interested here in the following more specific question.

Given a theoretically good parallel algorithm (e.g. divide-and-conquer matrix multiplication) and given a type of hardware that depends on various parameters (e.g. a GPGPU with Z words of private memory per SM and a maximum number M_{warp} of warps supported by an SM, etc.) we aim at automatically generating CUDA kernels that depends on

[1] In the MCM model, the number of SMs is unbounded as well as the size of the global memory, whereas each SM has a private memory of size Z.

[2] In the MCM model, the *work* of a thread-block is the total number of local operations (arithmetic operation, read/write memory accesses in the private memory of an SM) performed by all its threads; the *work* of a kernel is the sum of the works of all its thread-blocks.

[3] In the MCM model, the *span* of a thread-block is the maximum number of local operations performed by one of its threads; the *span* of a kernel is the maximum span among all its thread-blocks.

[4] In the MCM model, the *parallelism overhead* (or *overhead*, for short) of a thread-block accounts for the time to transfer data between the global memory of the machine and the private memory of the SM running this thread-block, taking coalesced accesses into account; the *parallelism overhead* of a kernel is the sum of the overheads of all its thread-blocks.

[5] Our algorithms are implemented in CUDA and publicly available with benchmarking scripts from `http://www.cumodp.org/`.

the hardware parameters (Z, M_{warp}, etc.), as well as program parameters (e.g. number ℓ of threads per block), such that

- those parameters need not to be known at compile-time, and
- are encoded as symbols in the generated kernel code.

For this reason, we call such CUDA kernels *parametric*.

In contrast, current technology requires that machine and program parameters are specialized to numerical values at the time of generating the GPGPU code, see [17, 3, 21, 33].

In order to clarify our objective, we briefly provide some background material. The *polyhedron model* [4] is a powerful geometrical tool for analyzing the relation (w.r.t. data locality or parallelization) between the iterations of nested for-loops. Once the polyhedron representing the *iteration space* of a loop nest is calculated, techniques of linear algebra and linear programming can transform it into another polyhedron encoding the loop steps into a coordinate system based on time and space (processors). From there, a parallel program can be generated. For example, for the following code computing the product of two univariate polynomials `a` and `b`, both of degree `n`, and writing the result to `c`,

```
for(i=0; i<=n; i++) {c[i] = 0; c[i+n] = 0;}
for(i=0; i<=n; i++) {
  for(j=0; j<=n; j++)
    c[i+j] += a[i] * b[j];
}
```

elementary dependence analysis suggests to set $t(i, j) = n-j$ and $p(i, j) = i + j$, where t and p represent time and processor respectively [16]. Using Fourier-Motzkin elimination, projecting all constraints on the (t, p)-plane yields the following asynchronous schedule of the above code:

```
parallel_for (p=0; p<=2*n; p++){
  c[p]=0;
  for (t=max(0,n-p); t<= min(n,2*n-p);t++)
    c[p] = c[p] + a[t+p-n] * b[n-t];
}
```

To be practically efficient, one should avoid a too fine-grained parallelization; this is achieved by grouping loop steps into so-called *tiles*, which are generally trapezoids [20]. It is also desirable for the generated code to depend on parameters such as tile and cache sizes, number of processors, etc. These extensions lead, however, to the manipulation of systems of non-linear polynomial equations and the use of techniques like quantifier elimination (QE). This was noticed by the authors of [16] who observed also that work remained to be done for adapting QE tools to the needs of automatic parallelization.

To illustrate these observations, we return to the above example and use a tiling approach: we consider a one-dimensional grid of thread-blocks where each block is in charge of updating at most B coefficients of the polynomial `c`. Therefore, we introduce three variables `B`, `b` and `u` where the latter two represent a thread-block index and an thread index (within a thread-block). This brings the following additional relations:

$$\left\{ \begin{array}{c} 0 \leq b \\ 0 \leq u < B \\ p = bB + u, \end{array} \right. \tag{1}$$

to the previous system

$$\left\{ \begin{array}{c} 0 < n \\ 0 \leq i \leq n \\ 0 \leq j \leq n \\ t = n - j \\ p = i + j. \end{array} \right. \tag{2}$$

To determine the target program, one needs to eliminate the variables `i` and `j`. In this case, Fourier-Motzkin elimination (FME) does not apply any more, due to the presence of non-linear constraints. If all the non-linear constraints appearing in a system of relations are polynomial constraints, the set of real solutions of such a system is a semi-algebraic set. The celebrated Tarski theorem [5] tells us that there always exists a quantifier elimination algorithm to project a semi-algebraic set of $\mathbb{R}^n$ to a semi-algebraic set of $\mathbb{R}^m$, $m \leq n$. The most popular method for conducting quantifier elimination (QE) of a semi-algebraic set is through cylindrical algebraic decomposition (CAD) [10]. Implementation of QE and CAD can be found in software such as QEPCAD, Reduce, MATHEMATICA as well as the `RegularChains` library of MAPLE [8]. Using the function QuantifierElimination (with options 'precondition'='AP', 'output'='rootof', 'simplification'='L4') in the `RegularChains` library, we obtain the following:

$$\left\{ \begin{array}{c} B > 0 \\ n > 0 \\ 0 \leq b \leq 2n/B \\ 0 \leq u < B \\ 0 \leq u \leq 2n - Bb \\ p = bB + u, \\ 0 \leq t \leq n, \\ n - p \leq t \leq 2n - p, \end{array} \right. \tag{3}$$

from where we derive the following program:

```
for (p=0; p<=2*n; p++) c[p]=0;
parallel_for (b=0; b<= 2 n / B; b++) {
    for (u=0; u<=min(B-1, 2*n - B * b); u++) {
       p = b * B + u;
       for (t=max(0,n-p); t<=min(n,2*n-p) ;t++)
           c[p] = c[p] + a[t+p-n] * b[n-t];
   }
}
```

An equivalent CUDA kernel to the parallel_for part is as below:

```
b = blockIdx.x;
u = threadIdx.x;
if (u <= 2*n - B * b) {
    p = b * B + u;
    for (t=max(0,n-p); t<=min(n,2*n-p) ;t++)
        c[p] = c[p] + a[t+p-n] * b[n-t];
}
```

We remark that the polynomial system defined by (1) and (2) has some special structure. The authors in [15] have exploited this structure to deduce a special algorithm to solve it and similar problems by implementing some parametric FME. Although the system (3) can be directly processed by QuantifierElimination, we found that it is much more efficient to use the following special QE procedure. We replace the product bB in system 1 by a new variable c, and

thus obtain a system of linear constraints. We then apply FME to eliminate the variables i, j, t, p, u in sequential. Now we obtain a system of linear constraints in variables c, b, n, B. Next we replace c by bB and have again a system of non-linear constraints in variables b, n, B. We then call QuantifierElimination to eliminate the variables b, n, B. The correctness of the procedure is easy to verify.

4. THE METAFORK LANGUAGE

In an earlier work [9], the second and the fourth authors of the present paper introduced MetaFork as an extension of both the C and C++ languages into a multithreaded language based on the fork-join concurrency model [7]. Thus, concurrent execution is obtained by a parent thread creating and launching one or more child threads so that the parent and its children execute a so-called *parallel region*. An important example of parallel regions are for-loop bodies. MetaFork has four parallel constructs dedicated to the fork-join model: function call spawn, block spawn, parallel for-loop and synchronization barrier. The first two use the keyword `meta_fork` while the other two use respectively the keywords `meta_for` and `meta_join`. Similarly to the CilkPlus specifications, the parallel constructs of MetaFork grant permission for concurrent execution but do not command it. Hence, a MetaFork program can execute on a single core machine. We emphasize the fact that `meta_fork` allows the programmer to spawn a function call (like in CilkPlus [6, 23, 22]) as well as a block (like in OpenMP [28, 2]).

Stencil computations are a major pattern in scientific computing. Stencil codes perform a sequence of sweeps (called time-steps) through a given array and each sweep can be seen as the execution of a pipeline. When expressed with concurrency platforms based (and limited) to the fork-join model, parallel stencil computations incur excessive parallelism overheads. This problem is studied in [29] together with a solution in the context of OpenMP by proposing new synchronization constructs to enable *do-across parallelism*. These observations have motivated a first extension of the MetaFork language with three constructs to express `pipelining` parallelism: `meta_pipe`, `meta_wait` and `meta_continue`. Recall that a pipeline is a linear sequence of processing stages through which data items flow from the first stage to the last stage. If each stage can process only one data item at a time, then the pipeline is said *serial* and can be depicted by a (directed) path in the sense of graph theory. If a stage can process more than one data item at a time, then the pipeline is said *parallel* and can be depicted by a directed acyclic graph (DAG), where each parallel stage is represented by an independent set, that is, a set of vertices of which no pair is adjacent. Since pipelining is not essential for the rest of the paper, we refer the interested reader to the MetaFork web site at `www.metafork.org`.

More recently, the MetaFork language was enhanced with constructs allowing the programmer to express the fact that a function call can be executed on an external (or remote) hardware component. This latter is referred as the *device* while the hardware component on which the MetaFork program was initially launched is refereed as the *host* and this program is then called the *host code*. Both the *host* and the *device* maintain their own separate memory spaces. Such function calls on an external device are expressed by means

```
void foo()
{
  int arry_host[N];
  initialize(arry_host, N);
  meta_fork bar(arry_host,
                        N);
  work();
}
```

```
void foo()
{
  int arry_host[N];
  initialize(arry_host, N);
  // declare an array on the device
  meta_device int arry_device[N];
  // copy 24 bytes of host array
  // from host to device
  meta_copy(arry_host+8, arry_host+32,
              arry_host, arry_device);
  meta_device bar(arry_device, N);
  work();
}
```

Figure 1: MetaFork **example**

of two new keywords: **meta_device** and **meta_copy**. We call a statement of the form

meta_device ⟨variable declaration⟩

a *device declaration*; it is used to express the fact that a variable is declared in the memory address space of the device.

A statement of the form

meta_device ⟨function call⟩

is called a *device function call*; it is used to express the fact that a function call is executed on the device concurrently (thus in a non-blocking way) to the execution of the parent thread on the host. All arguments in the function call must be either device-declared variables or values from primitive types (`char`, `float`, `int`, `double`).

A statement of the form

meta_copy (⟨range⟩, ⟨variable⟩, ⟨variable⟩)

copies the bytes whose memory addresses are in `range` (and who are assumed to be data referenced by the first variable) to the memory space referenced by the second variable[6]. Moreover, either one or both variables must be device-declared variables.

The left part of Figure 1 shows a MetaFork code fragment with a spawned function call operating on an array located in the shared memory of the host. On the right part of Figure 1, the same function is called on an array located in the memory of the device. In order for this function call to perform the same computation on the device as on the host, the necessary coefficients of the host array are copied to the device array. In this example, we assume that those coefficients are `arry_host[2]`, ..., `arry_host[8]`.

Several devices can be used within the same MetaFork program. In this case, each of the constructs above is followed by a number referring to the device to be used. Therefore, these function calls on an external device can be seen as one-sided message passing protocol, similar to the one of the `Julia` [7] programming language.

We stress the following facts about function calls on an external device. Any function declared in the host code can be invoked in a device function call. Moreover, within the body of a function invoked in a device function call, any other function defined in host code can be:

- either called (in the ordinary way, that is, as in the C language) and then executed on the same device,

[6]The difference between the lower end of the range and the memory address of the source array is used as offset to write in the second variable.

[7]Julia web site: `http://julialang.org/`

- or called on another device.

As a consequence, device function calls together with spawned function calls and ordinary function calls form a directed acyclic graphs of *tasks* to which the usual notions of the fork-join concurrency model can be applied.

The mechanism of function call on an external device can be used to model the distributed computing model of `Julia` as well as heterogeneous computing in the sense of CUDA. The latter is, however, more complex since, as in discussed in Section 2, it borrows from both the fork-join concurrency model and SIMD parallelism. In particular, a CUDA kernel call induces how the work is scheduled among the available SMs. Since our goal is to generate efficient CUDA code from an input MetaFork program, it is necessary to annotate MetaFork code in a more precise manner. To this end, we have introduced a tenth keyword, namely `meta_schedule`.

Any valid block of MetaFork code (like a `meta_for`-loop body) can be the body of `meta_schedule` statement. The semantic of that statement is that of its body and `meta_schedule` is an indication to the MetaFork-to-CUDA translator that every `meta_for`-loop nest of the `meta_schedule` statement must translate to a CUDA kernel call. If a `meta_for`-loop nest has 2 (resp. 4) nested loops then the outer one (resp. the two outermost ones) defines the grid of the kernel while the inner one (resp. the two innermost ones) specifies (resp. specify) the format of a thread-block. We skip the other possible configurations since they have not been implemented yet in the MetaFork compilation framework.

We conclude this section with an example, a one-dimensional stencil computation, namely Jacobi. The original (and naive) C version is shown below, where initialization statements have been removed in the interest of space.

```
for (int t = 0; t < T; ++t) {
    for (int i = 1; i < N-1; ++i)
        b[i] = (a[i-1] + a[i] + a[i+1]) / 3;

     for (int i = 1; i < N-1; ++i)
         a[i] = b[i];
}
```

From this C code fragment, we apply the tiling techniques mentioned in Section 3 and obtain the MetaFork code shown below. Observe that the `meta_schedule` statement has two `meta_for` loop nests, yielding two CUDA kernels.

```
int ub_v = (N - 2) / B;

meta_schedule {
    for (int t = 0; t < T; ++t) {
        meta_for (int v = 0; v < ub_v; v++) {
            meta_for (int u = 0; u < B; u++) {
               int p = v * B + u + 1;
               int y = p - 1;
               int z = p + 1;
               b[p] = (a[y] + a[p] + a[z]) / 3;
            }
        }
        meta_for (int v = 0; v < ub_v; v++) {
             meta_for (int u = 0; u < B; u++) {
                 int w = v * B + u + 1;
                 a[w] = b[w];
             }
         }
    }
}
```

From the above, our MetaFork-to-CUDA translator produces two kernel functions, a header file (for those two kernels) and a host code file where those kernels are called. The code of the first kernel is shown below. This code is generated automatically and correctly up to the `if` statement which requires manual post-processing in order to get the index arithmetic right. This limitation is already present on PPCG, the non-parametric CUDA code generator on which we rely. Fixing those "limit conditions" in stencil computation is work in progress.

```
#include "jacobi_kernel.hu"
__global__ void kernel0(int *a, int *b, int N,
              int T, int ub_v, int B, int c0)
{
    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    int private_p;
    int private_y;
    int private_z;
    extern __shared__ int shared_a[];

    #define floord(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
    for (int c1 = b0; c1 < ub_v; c1 += 32768) {

       if (!t0) {
        shared_a[(B)] = a[(c1 + 1) * (B)];
        shared_a[(B) + 1] = a[(c1 + 1) * (B) + 1];
      }
       if (N >= t0 + (B) * c1 + 1)
        shared_a[t0] = a[t0 + (B) * c1];
      __syncthreads();
      for (int c2 = t0; c2 < B; c2 += 512) {
        private_p = ((((c1) * (B)) + (c2)) + 1);
        private_y = (private_p - 1);
        private_z = (private_p + 1);
        b[private_p] = (((shared_a[private_y - (B) * c1] +
                          shared_a[private_p - (B) * c1]) +
                          shared_a[private_z - (B) * c1]) / 3);
      }
      __syncthreads();
    }
}
```

5. THE METAFORK GENERATOR OF PARAMETRIC CUDA KERNELS

In Section 3, we illustrated the process of parametric CUDA kernel generation from a sequential C program using MetaFork as an intermediate language. In this section, we assume that, from a C program, one has generated a MetaFork program which contains one or more `meta_schedule` blocks. Each such block contains parameters like thread block dimension sizes and is meant to be translated into a CUDA kernel.

For that latter task, we rely on PPCG [33], a C-to-CUDA code generator, that we have modified in order to generate parametric CUDA kernels. Figure 2 illustrates the software architecture of our C-to-CUDA code generator, based on PPCG. Since the original PPCG framework does not support parametric CUDA kernels the relevant data structures like non-linear expressions in the for-loop lower and upper bounds, are not supported either. Consequently, part of our code generation is done in a post-processing phase.

A `meta_schedule` block generating a one-dimensional grid with one-dimensional thread blocks has the following structure

```
meta_schedule {
  meta_for (int v = 0; v < grid_size; v++)
    meta_for (int u = 0; u < block_size; u++) {
      // Statements
    }
}
```

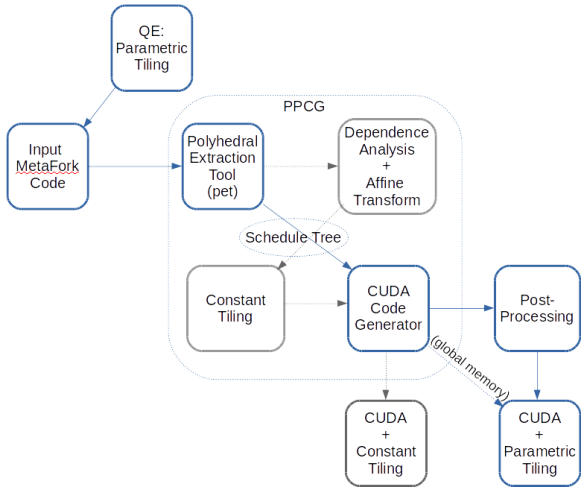


Figure 2: Overview of the implementation of MetaFork support for GPU code.

where the grid (resp. thread-block) dimension size is extracted from the outer (resp. inner) `meta_for` loop upper bound. Similarly, a `metaschedule` block generating a two-dimensional grid with one-dimensional thread block has the following structure

```
meta_schedule {
  meta_for (int v1 = 0; v1 < grid_size_y; v1++)
    meta_for (int v0 = 0; v0 < grid_size_x; v0++)
      // Only for-loops are valid here
      // For instance: for (int i = 0; i < n; i++)
      meta_for(int u1 = 0; u1 < block_size_y; u1++)
        meta_for(int u0 = 0; u0 < block_size_x; u0++)
        {
          // Statements
        }
}
```

where the first two outer `meta_for` loops correspond to the grid and the inner `meta_for` loops to the thread-blocks.

Among our adaptation of the PPCG framework, we have added a new node in the `PET` (Polyhedral Extraction Tool) phase of PPCG in order to represent the information of a `meta_for` loop; this is, in fact, very similar to a C for-loop except that we tag the outer `meta_for` loops as blocks and the inner `meta_for` loops as threads directly.

Since the MetaFork code is obtained after computing affine transformation and tiling (via quantifier elimination), we had to bypass the process of computing affine transformation and tiling that PPCG is performing. By doing this, our prototype C-to-CUDA code generator could not fully take advantage of PPCG, which explains why post-processing was necessary. Of course, improving this design is work in progress so as to completely avoid post-processing.

Referring to the above types of `meta_schedule` blocks, we use the "`block_size`" as the number of threads per thread block and "`grid_size`" as the number of thread blocks, we extract this information during the `PET` analysis and store it in the intermediate data structure, called 'schedule_tree', of PPCG. Hence, in the generated host code, the number of threads per block is specified as the variable: "`block_size`".

Originally, PPCG would first analyze whether an array is reused or accessed in a coalesced way, and then decide whether to use shared memory or not. In adaptation of the PPCG framework, we could not let PPCG analyze these properties of the input code and we simply force PPCG to consider using shared memory, provided that a corresponding compilation flag is enabled.

Since PPCG cannot deal with non-linear analysis, it cannot correctly handle the necessary index arithmetic formulas required by the shared memory accesses in our parametric code. Hence, this task is left to our post-processing phase. Our post-processing phase has another purpose: performing *common sub-expression elimination*. Indeed, our aromatically generated parametric CUDA kernel code requires that optimization, even more than PPCG does, due to the presence of parameters.

6. EXPERIMENTATION

In this section, we present experimental results. Most of them were obtained by running times of CUDA programs generated with

- our preliminary implementation of our MetaFork-to-CUDA code translator described in Sections 4 and 5, and
- the original version of the PPCG C-to-CUDA code translator [33].

We use eight simple examples: *array reversal* (Table 1), *1D Jacobi* (Table 2), *2D Jacobi* (Table 3), *LU decomposition* (Table 4), *matrix transposition* (Table 5), *matrix addition* (Table 6), *matrix vector multiplication* (Table 7), and *matrix matrix multiplication* (Table 8). In all cases, we use dense representations for our matrices and vectors.

For both the PPCG C-to-CUDA and our MetaFork-to-CUDA code translators, Tables 1, 2, 3, 4 5, 6, 7 and 8 give the speedup factors of the generated code, as the timing ratio of the generated code to their original untiled C code. Since PPCG determines a thread block format, the timings in those tables corresponding to PPCG depend only on the input data size. Meanwhile, since the CUDA kernels generated by MetaFork are parametric, the MetaFork timings are obtained for various formats of thread blocks and various input data sizes. Indeed, recall that our generated CUDA code admits parameters for the dimension sizes of the thread blocks. This generated parametric code is then specialized with the thread block formats listed in the first column of those tables.

Array reversal. Both MetaFork and PPCG generate CUDA code that uses shared array and one-dimensional kernel grid. We specialized the MetaFork generated parametric code successively to the block size $B = 16, 32, 64, 128, 256, 512$, meanwhile PPCG automatically chooses $B = 32$ as the block size. As we can see in Table 1, based on the generated parametric CUDA kernel, one can tune the block size to be 256 to obtain the best performance.

1D Jacobi. Our second example is a one-dimensional stencil computation, namely 1D-Jacobi. The kernel generated by MetaFork uses a 1D kernel grid and share array while the kernel generated by PPCG uses a 1D kernel grid and global memory. PPCG automatically chooses a thread block format of 32 while MetaFork preferred format varies based on input size.

2D Jacobi. Our next example is a two-dimensional stencil computation, namely 2D-Jacobi. Both the CUDA kernels

generated by MetaFork and PPCG use a 2D kernel grid and global memory. PPCG automatically chooses a thread block format of 16×32 while MetaFork preferred format is 4×32.

LU decomposition. MetaFork and PPCG both generate two CUDA kernels: one with a 1D grid and one with a 2D grid, both using shared memory. The automatically selected block formats for PPCG are 32 and 16×32. Meanwhile, tuning the number of threads per thread block in our parametric code allows MetaFork to outperform PPCG.

Matrix transpose. Both the CUDA kernels generated by MetaFork and PPCG use a 2D grid and shared memory. PPCG automatically chooses a thread block format of 16×32 while MetaFork preferred format varies based on input size. The MetaFork-generated code is about twice slower than the PPCG-generated code. This is due to the fact that PPCG manages to optimize the generated code so as ensure coalesced accesses. Once MetaFork and PPCG are better integrated together, the MetaFork-generated code should also benefit from this optimization performed by PPCG.

Matrix addition. Both the CUDA kernels generated by MetaFork and PPCG use a 2D grid and global memory. Hence, they fail to generate shared memory code. The automatically chosen block format for PPCG is 16×32 while MetaFork preferred format is 4×32. In this case, the fact that MetaFork uses common sub-expression elimination in its post-processing phase explains why MetaFork outperforms PPCG by one order of magnitude.

Speedup (kernel)	Input size			
Block size	2^{23}	2^{24}	2^{25}	2^{26}
PPCG				
32	8.312	8.121	8.204	8.040
MetaFork				
16	3.558	3.666	3.450	3.445
32	7.107	6.983	7.039	6.831
64	12.227	12.591	12.763	12.782
128	17.743	19.506	19.733	19.952
256	**19.035**	**21.235**	**22.416**	**21.841**
512	18.127	18.017	19.206	20.587

Table 1: Reversing a one-dimensional array

Speedup (kernel)	Input size		
Block size	2^{13}	2^{14}	2^{15}
PPCG			
32	1.416	2.424	5.035
MetaFork			
16	1.217	1.890	2.954
32	1.718	2.653	5.059
64	1.679	3.222	7.767
128	1.819	3.325	**10.127**
256	1.767	**3.562**	10.077
512	**2.081**	3.161	9.654

Table 2: 1D-Jacobi

Matrix vector multiplication. For both MetaFork and PPCG, the generated kernels use a 1D grid and shared memory. The block size chosen by PPCG is 32 while MetaFork preferred block size is 128.

Matrix matrix multiplication. For both MetaFork and PPCG, the generated kernels use a 2D grid and shared memory. On the contrary of matrix vector multiplication, both

Speedup (kernel)	Input size		
Block size	2^{12}	2^{13}	2^{14}
PPCG			
16 * 32	8.614	8.672	10.6316
MetaFork			
4 * 8	4.293	4.209	3.539
4 * 16	7.091	6.696	6.029
4 * 32	**9.116**	**8.388**	**7.916**
8 * 8	5.103	4.785	4.300
8 * 16	7.340	6.173	5.203
8 * 32	8.227	7.375	6.541
16 * 8	4.286	3.934	3.172
16 * 16	5.375	4.862	3.904
16 * 32	5.479	5.196	4.771

Table 3: 2D-Jacobi

Speedup (kernel)	Input size	
Block size `kernel0`, `kernel1`	2^{12}	2^{13}
PPCG		
32, 16 * 32	31.497	39.068
MetaFork		
32, 4 * 4	18.906	27.025
64, 4 * 4	18.763	27.316
128, 4 * 4	18.713	27.109
256, 4 * 4	18.553	27.259
512, 4 * 4	18.607	27.353
32, 8 * 8	**34.936**	52.850
64, 8 * 8	34.163	**53.133**
128, 8 * 8	34.050	52.731
256, 8 * 8	33.932	52.616
512, 8 * 8	34.850	53.112
32, 16 * 16	32.310	41.131
64, 16 * 16	32.093	40.829
128, 16 * 16	32.968	41.219
256, 16 * 16	32.229	41.246
512, 16 * 16	32.806	40.705

Table 4: LU decomposition

MetaFork and PPCG could generate a blocked multiplication which explains the better performances. For MetaFork, the size of the shared array is the same as block format. However, for PPCG, the size of the shared array is 32×32 while the thread-block format is 16×32. In fact, PPCG code computes two coefficients of the output matrix with each thread, thus increasing index arithmetic amortization and occupancy. This is another optimization that could benefit to MetaFork once MetaFork and PPCG are better integrated together,.

We conclude this section with timings (in seconds) for the quantifier elimination (QE) required to generate MetaFork tiled code, see Table 9. Our tests are based on the latest version of the `RegularChains` library of Maple, available at `www.regularchains.org`. These results show that the use of QE is not a bottleneck in our C-to-CUDA code generation process, despite of the theoretically high algebraic complexity of quantifier elimination.

7. CONCLUDING REMARKS

MetaFork can be applied for (1) comparing algorithms written with different concurrency platforms and (2) porting more programs to systems that may have a highly optimized run-time for one paradigm (say divide-and-conquer algorithms, or producer-consumer). These features have been illustrated in [9].

Speedup (kernel)	Input size		
Block size	2^{12}	2^{13}	2^{14}
PPCG			
16 * 32	63.656	62.656	103.703
MetaFork			
4 * 8	16.729	19.887	27.368
4 * 16	23.755	27.653	**46.240**
4 * 32	29.460	**34.353**	41.814
8 * 8	18.725	18.583	20.408
8 * 16	**29.974**	32.746	38.575
8 * 32	24.798	29.043	37.027
16 * 8	15.664	14.383	16.949
16 * 16	16.087	20.341	26.160
16 * 32	15.133	16.707	19.591

Table 5: Matrix transpose

Speedup (kernel)	Input size	
Block size	2^{12}	2^{13}
PPCG		
16 * 32	11.352	9.679
MetaFork		
4 * 8	32.502	32.711
4 * 16	53.730	56.774
4 * 32	**65.975**	**57.395**
8 * 8	47.729	45.603
8 * 16	48.844	55.834
8 * 32	51.512	44.619
16 * 8	41.956	36.952
16 * 16	46.361	27.761
16 * 32	40.376	24.024

Table 6: Matrix addition

In this new paper, we have presented enhancements of the MetaFork language so as to provide several models of concurrency, including pipelining and SIMD. For the latter, our objective is to facilitate automatic code translation from high-level programming models supporting hardware accelerator (like OpenMP and OpenACC) to low-level heterogeneous programming models (like CUDA),

We believe that an important feature of MetaFork is the abstraction level that it provides. It can be useful for parallel language design (for example in designing parallel extensions to C/C++) as well as a good tool to teach parallel programming. At the same time, as illustrated in Sections 5 through 6, MetaFork has language constructs to help generating efficient CUDA code. Moreover, the MetaFork framework relies on advanced techniques (quantifier elimination in non-linear polynomial expressions) for code optimization, in particular tiling.

The experimentation reported in Section 6 show the benefits of generating *parametric* CUDA kernels. Not only this feature provides more portability but it helps obtaining better performance with automatically generated code.

Speedup (kernel)	Input size		
Block size	2^{11}	2^{12}	2^{13}
PPCG			
32	3.954	3.977	5.270
MetaFork			
16	9.343	11.752	22.486
32	20.494	17.847	22.486
64	21.360	42.008	38.858
128	23.141	**47.759**	**75.857**
256	**25.092**	44.166	73.447
512	22.613	47.641	62.922

Table 7: Matrix vector multiplication

Speedup (kernel)	Input size	
Block size	2^{10}	2^{11}
PPCG		
16 * 32	129.853	393.851
MetaFork		
4 * 8	22.620	80.610
4 * 16	39.639	142.244
4 * 32	37.372	135.583
8 * 8	**48.463**	**172.871**
8 * 16	43.720	162.263
8 * 32	33.071	122.960
16 * 8	30.128	101.367
16 * 16	34.619	133.497
16 * 32	22.600	84.319

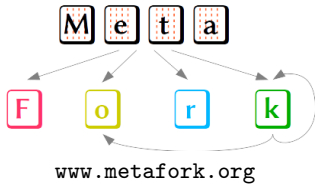
Table 8: Matrix multiplication

Example	Timing
Array reversal	0.072
1D Jacobi	0.948
2D Jacobi	7.735
LU decomposition	4.416
matrix transposition	1.314
matrix addition	1.314
matrix vector multiplication	0.072
matrix matrix multiplication	2.849

Table 9: Timings of quantifier elimination

Acknowledgements.

This work was supported in the by the IBM CAS Fellowship # 880 *Code Optimization Tools for Hardware Acceleration Technologies*, NSERC of Canada and the NSFC grants 11301524 and 11471307.



References

[1] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1):3–28, 1990.

[2] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP Tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418, 2009.

[3] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, CC'10/ETAPS'10, pages 244–263, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and*

Compilation Techniques, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[5] S. Basu, R. Pollack, and M.-F. Roy. *Algorithms in real algebraic geometry*, volume 10 of *Algorithms and Computations in Mathematics*. Springer-Verlag, 2006.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[7] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. Comput.*, 27(1):202–229, 1998.

[8] C. Chen and M. Moreno Maza. Quantifier elimination by cylindrical algebraic decomposition based on regular chains. In *International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014*, pages 91–98, 2014.

[9] X. Chen, M. Moreno Maza, S. Shekar, and P. Unnikrishnan. Metafork: A framework for concurrency platforms targeting multicores. In *Using and Improving OpenMP for Devices, Tasks, and More - 10th International Workshop on OpenMP, IWOMP 2014, Brazil, September 28-30, 2014. Proceedings*, pages 30–44, 2014.

[10] G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. *Springer Lecture Notes in Computer Science*, 33:515–532, 1975.

[11] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965.

[12] P. B. Gibbons. A more practical PRAM model. In *Proc. of SPAA*, pages 158–168. ACM, 1989.

[13] P. B. Gibbons, Y. Matias, and V. Ramachandran. The Queue-Read Queue-Write PRAM model: Accounting for contention in parallel algorithms. *SIAM J. on Comput.*, 28(2):733–769, 1998.

[14] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. on Applied Mathematics*, 17(2):416–429, 1969.

[15] A. Größlinger, M. Griebl, and C. Lengauer. Introducing non-linear parameters to the polyhedron model. In *Proceedings of 11th Workshop on Compilers for Parallel Computers*, CPC '04, pages 1–12, 2004.

[16] A. Größlinger, M. Griebl, and C. Lengauer. Quantifier elimination in automatic loop parallelization. *J. Symb. Comput.*, 41(11):1206–1221, 2006.

[17] T. D. Han and T. S. Abdelrahman. ThiCUDA: a high-level directive based language for GPU programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units,*, GPGPU-2, pages 52–61, New York, NY, USA, 2009.

[18] S. A. Haque, M. Moreno Maza, and N. Xie. A many-core machine model for designing algorithms with minimum parallelism overheads. In *Proc. of International Conference on Parallel Computing (ParCo)*, 2015.

[19] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *Proc. of SPAA*, pages 145–156, 2010.

[20] K. Högstedt, L. Carter, and J. Ferrante. Determining the idle time of a tiling. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 160–173, New York, NY, USA, 1997. ACM.

[21] J. Holewinski, L. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 311–320, New York, NY, USA, 2012. ACM.

[22] Intel Corporation. Intel CilkPlus language specification, version 0.9, 2013.

[23] C. E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[24] L. Ma, R. D. Chamberlain, and K. Agrawal. Performance modeling for highly-threaded many-core GPUs. In *IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2014, Zurich, Switzerland, June 18-20, 2014*, pages 84–91. IEEE, 2014.

[25] L. Meng, Y. Voronenko, J. R. Johnson, M. Moreno Maza, F. Franchetti, and Y. Xie. Spiral-generated modular FFT algorithms. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 169–170, USA, 2010. ACM.

[26] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[27] NVIDIA. NVIDIA next generation CUDA compute architecture: Kepler GK110, 2012.

[28] OpenMP Architecture Review Board. OpenMP application program interface, version 4.0, 2013.

[29] J. Shirako, P. Unnikrishnan, S. Chatterjee, K. Li, and V. Sarkar. Expressing doacross loop dependences in OpenMP. In *IWOMP*, volume 8122 of *Lecture Notes in Computer Science*, pages 30–44. Springer, 2013.

[30] T. G. Jr. Stockham. High-speed convolution and correlation. In *Proc. of AFIPS*, pages 229–233. ACM, 1966.

[31] L. J. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM J. Comput.*, 13(2):409–422, 1984.

[32] J. E Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[33] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for CUDA. *TACO*, 9(4):54, 2013.