Comprehensive Optimization of Parametric Kernels for Graphics Processing Units

Xiaohui Chen
AMD
Markham, ON
Canada

Marc Moreno Maza
U. Western Ontario
London, ON
Canada

Jeeva Paudel
IBM Canada Ltd
Markham, ON
Canada

Ning Xie
Huawei Technologies Canada Ltd
Markham, ON
Canada

January 16, 2018

Abstract

This work deals with the optimization of computer programs targeting Graphics Processing Units (GPUs). The goal is to lift, from programmers to optimizing compilers, the heavy burden of determining program details that are dependent on the hardware characteristics. The expected benefit is to improve robustness, portability and efficiency of the generated computer programs. We address these requirements by:

• treating machine and program parameters as unknown symbols during code generation, and
• generating optimized programs in the form of a case discussion, based on the possible values of the machine and program parameters.

By taking advantage of recent advances in the area of computer algebra, preliminary experimentation yield promising results.

1 Introduction

It is well-known that the advent of hardware acceleration technologies (multicore processors, graphics processing units, field programmable gate arrays) provide vast opportunities for innovation in computing. In particular, GPUs combined with low-level heterogeneous programming models, such as CUDA (the Compute Unified Device Architecture, see [19, 2]), brought super-computing to the level of the desktop computer. However, these low-level programming models carry notable challenges, even to expert programmers. Indeed, fully exploiting the power of hardware accelerators by writing CUDA code often requires significant code optimization effort. While such effort can yield high performance, it is desirable for many programmers to avoid the explicit management of the hardware accelerator, e.g. data transfer between host and device, or between memory levels of the device. To this end, high-level models for accelerator programming, notably OPENMP [12, 4] and OPENACC [24, 3], have become an important research direction. With these models, programmers only need to annotate their C/C++ (or FORTRAN) code to indicate which portion of code is to be executed on the device, and how data is mapped between host and device.

In OPENMP and OPENACC, the division of the work between thread blocks within a grid, or between threads within a thread block, can be expressed in a loose manner, or even ignored. This implies that code optimization techniques must be applied in order to derive efficient CUDA code. Moreover, existing software packages (e.g. PPCG [25], C-TO-CUDA [7], HiCUDA [15], CUDA-CHILL [16]) for generating CUDA code from annotated C/C++ programs, either let the user choose, or make assumptions on, the characteristics of the targeted hardware, and on how the work is divided among the processors of that device. These choices and assumptions limit code portability as well as opportunities for code optimization.
optimization. This latter fact will be illustrated with dense matrix multiplication, through Figures 3 and 4 as well as Table 1.

To deal with these challenges in translating annotated C/C++ programs to CUDA, we propose in [9] to generate parametric CUDA kernels, that is, CUDA kernels for which program parameters (e.g. number of threads per thread block) and machine parameters (e.g. shared memory size) are symbolic entities instead of numerical values. Hence, the values of these parameters need not to be known during code generation: machine parameters can be looked up when the generated code is loaded on the target machine, while program parameters can be deduced, for instance, by auto-tuning. See Figure 4 for an example of parametric CUDA kernels. A proof-of-concept implementation, presented in [9] and publicly available1 uses another high-level model for accelerator programming, called METAORK, that we introduced in [11]. The experimentation shows that the generation of parametric CUDA kernels can lead to significant performance improvement w.r.t. approaches based on the generation of CUDA kernels that are not parametric. Moreover, for certain test-cases, our experimental results show that the optimal choice for program parameters may depend on the input data size. For instance, the timings gathered in Table 1 show that the format of the 2D thread-block of the best CUDA kernel that we could generate is $16 \times 8$ for matrices of order $2^{10}$ and $32 \times 8$ for matrices of order $2^{11}$. Clearly, parametric CUDA kernels are well-suited for this type of test-cases.

In this paper, our goal is to enhance the framework initiated in [9] by generating optimized parametric CUDA kernels. As we shall see, this can be done in the form of a case discussion, based on the possible values of the machine and program parameters. The output of a procedure generating optimized parametric CUDA kernels will be called a comprehensive parametric CUDA kernel. A simple example is shown on Figure 2. In broad terms, this is a decision tree where:

1. each internal node is a Boolean condition on the machine and program parameters, and
2. each leaf is a CUDA program $P$, optimized w.r.t. prescribed criteria and optimization techniques, under the conjunction of the conditions along the path from the root of the tree to $P$.

The intention, with this concept, is to automatically generate optimized CUDA kernels from annotated C/C++ code without knowing the numerical values of some or even any of the machine and program parameters. This naturally leads to case distinction depending on the values of those parameters, which materializes into a disjunction of conjunctive non-linear polynomial constraints. Symbolic computation, aka computer algebra, is the natural framework for manipulating such systems of constraints; our RegularChains library2 provides the appropriate algorithmic tools for that task.

Other research groups have approached the questions of code portability and code optimization in the context of CUDA code generation from high-level programming models. They use techniques like auto-tuning [14, 16], dynamic instrumentation [17] or both [22]. Rephrasing [16], “those techniques explore empirically different data placement and thread/block mapping strategies, along with other code generation decisions, thus facilitating the finding of a high-performance solution.”

In the case of auto-tuning techniques, which have been used successfully in the celebrated projects ATLAS [27], FFTW [13], and SPiRAL [20], part of the code optimization process is done off-line, that is, the input code is analyzed and an optimization strategy (i.e. a sequence of composable code transformations) is generated, and then applied on-line (i.e. on the targeted hardware). We propose to push this idea further by applying the optimization strategy off-line, thus, even before the code is loaded on the targeted hardware.

Let us illustrate, with an example, the notion of comprehensive parametric CUDA kernels, along with a procedure to generate them. Our input is the for-loop nest of Figure 1 which computes the sum of two matrices $b$ and $c$ of order $N$ using a blocking strategy; each matrix is divided into blocks of format $B_0 \times B_1$. This input code is annotated for parallel execution in the METAORK language. The body of the

---

1. www.metafork.org
2. This library, shipped with the commercialized computer algebra system MAPLE, is freely available at www.regularchains.org.
statement `meta_schedule` is meant to be offloaded to a GPU device and each `meta_for` loop is a parallel for-loop where all iterations can be executed concurrently.

```c
int dim0 = N/B0, dim1 = N/(2*B1);
meta_schedule {
    meta_for (int v = 0; v < dim0; v++)
    meta_for (int p = 0; p < dim1; p++)
    meta_for (int u = 0; u < B0; u++)
    meta_for (int q = 0; q < B1; q++) {
        int i = v * B0 + u;
        int j = p * B1 + q;
        if (i < N && j < N/2) {
            c[i][j] = a[i][j] + b[i][j];
            c[i][j+N/2] = a[i][j+N/2] + b[i][j+N/2];
        }
    }
}
```

Figure 1: A `meta_for` loop nest for adding two matrices.

We make the following simplistic assumptions for the translation of this for-loop nest to CUDA.
1. The target machine has two parameters: the maximum number $R$ of registers per thread, and the maximum number $T$ of threads per thread-block; all other hardware limits are ignored.
2. The generated kernels depend on two program parameters, $B_0$ and $B_1$, which define the format of a 2D thread-block.
3. The optimization strategy (w.r.t. register usage per thread) consists in reducing the work per thread (by reducing loop granularity).

A possible comprehensive parametric CUDA kernel is given by the pairs $(C_1, K_1)$ and $(C_2, K_2)$, where $C_1, C_2$ are two sets of algebraic constraints on the parameters and $K_1, K_2$ are two CUDA kernels that are optimized under the constraints respectively given by $C_1, C_2$, see Figure 2. The following computational steps yield the pairs $(C_1, K_1)$ and $(C_2, K_2)$.

(S1) The METAORK code is mapped to an intermediate representation (IR) say that of LLVM or alternatively, to PTX code.
(S2) Using this IR (or PTX) code, one estimates the number of registers that a thread requires; on this example, using LLVM IR, we obtain an estimate of 14.
(S3) Next, we apply the optimization strategy, yielding a new IR (or PTX) code, for which register pressure reduces to 10. Since no other optimization techniques are considered, the procedure stops with the result shown on Figure 2.

Note that, strictly speaking, the kernels $K_1$ and $K_2$ on Figure 2 should be given by PTX code. But for simplicity, we are presenting them by counterpart CUDA code.

This paper is organized as follows. In Section 2, we review the notion of a parametric CUDA kernel through an example. In Section 3, we introduce the notion of comprehensive optimization of a code fragment together with an algorithm for computing it. In Section 4, we explain how this latter notion applies to the generation of parametric CUDA kernels generated from a program written in a high-level accelerator model namely METAORK. Finally, experimental results are provided in Section 5.

---

3 Quoting Wikipedia: “The LLVM compiler infrastructure project (formerly Low Level Virtual Machine [18]) is a framework for developing compiler front ends and back ends”.

4 The Parallel Thread Execution (PTX) is the pseudo-assembly language to which CUDA programs are compiled by NVIDIA’s NVCC compiler. PTX code can also be generated from (enhanced) LLVM IR, using nvptx back-end, following the work of [21].
2 Parametric kernels

We review and illustrate the notion of a parametric CUDA kernel (introduced in [9]) with an example: computing the product of two dense square matrices of order $n$. Figure 3 shows a code snippet, expressed in the METAFORK language, performing a blocking strategy. Each iteration of the parallel for-loop nest (i.e. the 4 $\text{meta_for}$ nested loops) computes $s$ coefficients of the product matrix. The blocks in the matrices $a, b, c$ have format $B_0 \times B_0$, $B_0 \times (ub1 \cdot s)$, $B_0 \times (ub1 \cdot s)$. Note that memory accesses to $a, b, c$ are coalesced in both codes.

Figure 4 shows a CUDA kernel code generated from the METAFORK code snippet of Figure 3. Observe that kernel0 takes the program parameters $B_0$ and $ub1$ as arguments, whereas non-parametric CUDA kernels usually only take data parameters (here $a, b, c, n$) as input arguments. Note also that, in order to allocate memory for the shared arrays shared_a, shared_b, shared_c, we predefine the names $B_0, B_1$ as macros and specify their values at compile time. Note that the assert statements ensure that $B_0, ub1$ match $B_0, B_1$.

To conclude with this example, we gather in Table 1 speedup factors w.r.t. a highly optimized serial C program implementing the same blocking strategy. The numbers in bold fonts correspond to the best speedup factors by a parametric kernel on a given input size. We observe that:

1. for $s = 4, \ ub1 = 16, \ B0 = 8$ when $n = 2^{10}$, and
2. for $s = 4, \ ub1 = 32, \ B0 = 8$ when $n = 2^{11}$, the parametric kernel of Figure 4 provides the best results.
assert(B0 <= ub1 * s);
int dim0 = n / B0, dim1 = n / (ub1 * s);
meta_schedule {
    meta_for (int i = 0; i < dim0; i++)
    meta_for (int j = 0; j < dim1; j++)
    for (int k = 0; k < n / B0; k++)
        meta_for (int v = 0; v < B0; v++)
            meta_for (int u = 0; u < ub1; u++) {
                int p = i * B0 + v;
                int q = j * ub1 * s + u;
                for (int z = 0; z < B0; z++)
                    for (int w = 0; w < s; w++) {
                        int x = w * ub1;
                        c[p][q+x] += a[p][B0*k+z] * b[B0*k+z][q+x];
                    }
            }
}
}

Figure 3: METAFORK matrix multiplication using a blocking strategy in METAFORK.

| Thread-block \ Input | $2^{10} \times 2^{10}$ | $2^{11} \times 2^{11}$ |
|----------------------|----------------------|----------------------|
| (ub1, B0)            | s = 2                | s = 4                |
| (16, 4)              | 95                   | 128                  |
| (32, 4)              | 128                  | 157                  |
| (64, 4)              | 111                  | 145                  |
| (8, 8)               | 131                  | 151                  |
| (16, 8)              | 164                  | 194                  |
| (32, 8)              | 163                  | 187                  |
| (64, 8)              | 94                   | 143                  |
| B0                   | Register usage for s = 4 |
| 4                    | 38                   |
| 8                    | 34                   |

Table 1: Speedup factors on an NVIDIA Tesla M2050 for our kernel generated by METAFORK with compilation flag -maxrregcount=40.

3 Comprehensive Optimization

We consider a code fragment written in one of the linguistic extensions of the C language targeting a computer device, for instance, a hardware accelerator. We assume that some, or all, of the hardware characteristics of this device are unknown at compile time. However, we would like to optimize our input code fragment w.r.t prescribed resource counters (e.g. memory usage) and performance counters (e.g. occupancy on a GPU device). To this end, the hardware characteristics of this device, as well as the program and data parameters of the code fragment, are treated as symbols. From there, we generate polynomial constraints (with those symbols as indeterminate variables) so as to (i) ensure that sufficient resources are available to run the transformed code, and (ii) attempt to improve the code performance.

3.1 Hypotheses on the input code fragment

We consider a sequence $\mathcal{S}$ of statements from the C programming language and introduce the following.

**Definition 1** We call parameter of $\mathcal{S}$ any scalar variable that is (i) read in $\mathcal{S}$ at least once, and (ii) never written in $\mathcal{S}$. We call data of $\mathcal{S}$ any non-scalar variable (e.g. array) that is not initialized but
__global__ void kernel0(int *a, int *b, int *c, int n, int dim0, int dim1, int B0, int ub1, int s) {
    int b0 = blockIdx.y, b1 = blockIdx.x;
    int t0 = threadIdx.y, t1 = threadIdx.x;
    int private_p, private_q;
    assert(B_0 == B0); assert(B_1 == ub1 * s);
    __shared__ int shared_a[B0][B0];
    __shared__ int shared_b[B0][B1];
    int private_c[1][S]; assert(S == s);
    for (int c0 = b0; c0 < dim0; c0 += 256)
        for (int c1 = b1; c1 < dim1; c1 += 256) {
            private_p = ((c0) * (B0)) + (t0);
            private_q = ((c1) * (ub1 * s)) + (t1);
            for (int c5 = 0; c5 < S; c5 += 1)
                if (n >= private_p + 1 &&
                    n >= private_q + (c5) * (ub1) + 1)
                    private_c[0][c5] = c[(private_p) * n +
                        (private_q + (c5) * (ub1))];
            for (int c2 = 0; c2 < n / B0; c2 += 1) {
                if (t1 < B0 && n >= private_p + 1)
                    shared_a[t0][t1] =
                        a[(private_p) * n + (t1 + B0 * c2)];
            for (int c5 = 0; c5 < S; c5 += 1)
                if (t0 < B0 &&
                    n >= private_q + (c5) * (ub1) + 1)
                    shared_b[t0][c5] = b[(t0 + B0 * c2) * n +
                        (private_q + (c5) * (ub1))];
            __syncthreads();
            for (int c6 = 0; c6 < B0; c6 += 1)
                if (c5 < S; c5 += 1)
                    private_c[0][c5] =
                        shared_a[t0][c6] *
                        shared_b[c6][c5 * ub1 + t1]);
            __syncthreads();
        }
        for (int c5 = 0; c5 < S; c5 += 1)
            if (n >= private_p + 1 &&
                n >= private_q + (c5) * (ub1) + 1)
                c[(private_p) * n +
                    (private_q + (c5) * (ub1))] =
                    private_c[0][c5];
            __syncthreads();
    }
}

Figure 4: CUDA kernel generated from a meta_schedule statement in the METAFORK language.

possibly overwritten within \( S \). If a parameter of \( S \) gives a dimension size of a data of \( S \), then this parameter is called a data parameter; otherwise, it is simply called a program parameter.

We denote by \( D_1, \ldots, D_u \) and \( E_1, \ldots, E_v \) the data parameters and program parameters of \( S \), respectively.

We make the following assumptions on \( S \):

(H1) All parameters are assumed to be non-negative integers.

(H2) We assume that \( S \) can be viewed as the body of a valid C function having the parameters and data of \( S \) as unique arguments.

The sequence of statements \( S \) can be the body of a kernel function in CUDA. In the kernel code of Figure 4, \( B0 \) and \( ub1 \) are program parameters while \( a \), \( b \) and \( c \) are the data, and that \( n \) is the data parameter.
3.2 Hardware resource limits and performance measures

We denote by \( R_1, \ldots, R_s \) the hardware resource limits of the targeted hardware device. Examples of these quantities for the NVIDIA Kepler micro-architecture are the maximum number of registers to be allocated per thread, and the maximum number of shared memory words to be allocated per thread-block. We denote by \( P_1, \ldots, P_t \) the performance measures of a program running on the device. These are dimensionless quantities defined as percentages. Examples of these quantities for the NVIDIA Kepler micro-architecture are the SM occupancy (that is, the ratio of the number of active warps to the maximum number of active warps) and the cache hit rate in an streaming multi-processor (SM).

For a given hardware device, \( R_1, \ldots, R_s \) are positive integers, and each of them is the maximum value of a hardware resource. Meanwhile, \( P_1, \ldots, P_t \) are rational numbers between 0 and 1. However, for the purpose of writing code portable across a variety of devices with similar characteristics, the quantities \( R_1, \ldots, R_s \) and \( P_1, \ldots, P_t \) will be treated as unknown and independent variables. These hardware resource limits and performance measures will be called the machine parameters.

Each function \( K \) (and, in particular, our input code fragment \( \mathcal{S} \)) written in the C language for the targeted hardware device has resource counters \( r_1, \ldots, r_s \) and performance counters \( p_1, \ldots, p_t \) corresponding, respectively, to \( R_1, \ldots, R_s \) and \( P_1, \ldots, P_t \). In other words, the quantities \( r_1, \ldots, r_s \) are the amounts of resources, corresponding to \( R_1, \ldots, R_s \), respectively, that \( K \) requires for executing. Similarly, the quantities \( p_1, \ldots, p_t \) are the performance measures, corresponding to \( P_1, \ldots, P_t \), respectively, that \( K \) exhibits when executing. Therefore, the inequalities \( 0 \leq r_1 \leq R_1, \ldots, 0 \leq r_s \leq R_s \) must hold for the function \( K \) to execute correctly. Similarly, \( 0 \leq p_1 \leq 1, \ldots, 0 \leq p_t \leq 1 \) are satisfied by the definition of the performance measures.

Remark 1 We note that \( r_1, \ldots, r_s, p_1, \ldots, p_t \) may be numerical values, which we can assume to be non-negative rational numbers. This will be the case, for instance, for the minimum number of registers required per thread in a thread-block. The resource counters \( r_1, \ldots, r_s \) may also be polynomial expressions whose indeterminate variables can be program parameters (like the dimension sizes of a thread-block or grid) or data parameters (like the input data sizes). Meanwhile, the performance counters \( p_1, \ldots, p_t \) may further depend on the hardware resource limits (like the maximum number of active warps supported by an SM). To summarize, we observe that \( r_1, \ldots, r_s \) are polynomials in \( \mathbb{Q}[D_1, \ldots, D_u, E_1, \ldots, E_v] \) and \( p_1, \ldots, p_t \) are rational functions where numerators and denominators are in \( \mathbb{Q}[D_1, \ldots, D_u, E_1, \ldots, E_v, R_1, \ldots, R_s] \).

Moreover, we can assume that the denominators of those functions are positive.

On Figure 2, \( R \) and \( T \) are machine parameters while \( B_0 \) and \( B_1 \) are program parameters. The constraints displayed on Figure 2 are polynomials in \( R, T, B_0, B_1 \).

3.3 Evaluation of resource and performance counters

Let \( G_C(\mathcal{S}) \) be the control flow graph (CFG) of \( \mathcal{S} \). Hence, the statements in the basic blocks of \( G_C(\mathcal{S}) \) are C statements, and we call such a CFG the source CFG. We also map \( \mathcal{S} \) to an intermediate representation, which, itself, is encoded in the form of a CFG, denoted by \( G_L(\mathcal{S}) \), and we call it the IR CFG. Here, we refer to the landmark textbook [6] for the notion of the control flow graph and that of intermediate representation.

We observe that \( \mathcal{S} \) can trivially be reconstructed from \( G_C(\mathcal{S}) \); hence, the knowledge of \( \mathcal{S} \) and that of \( G_C(\mathcal{S}) \) can be regarded as equivalent. In contrast, \( G_L(\mathcal{S}) \) depends not only on \( \mathcal{S} \) but also on the optimization strategies that are applied to the IR of \( \mathcal{S} \).

Equipped with \( G_C(\mathcal{S}) \) and \( G_L(\mathcal{S}) \), we assume that we can estimate each of the resource counters \( r_1, \ldots, r_s \) (resp. performance counters \( p_1, \ldots, p_t \)) by applying functions \( f_1, \ldots, f_s \) (resp. \( g_1, \ldots, g_t \)) to either \( G_C(\mathcal{S}) \) or \( G_L(\mathcal{S}) \). We call \( f_1, \ldots, f_s \) (resp. \( g_1, \ldots, g_t \)) the resource (resp. performance) evaluation functions.
For instance, when $S$ is the body of a CUDA kernel and $\mathcal{S}$ reads (resp. writes) a given array, computing the total amount of elements read (resp. written) by one thread-block can be determined from $G_C(\mathcal{S})$. Meanwhile, computing the minimum number of registers to be allocated to a thread executing $\mathcal{S}$ requires the knowledge of $G_L(\mathcal{S})$.

### 3.4 Optimization strategies

In order to reduce the consumption of hardware resources and increase performance counters, we assume that we have optimization procedures $O_1, \ldots, O_w$, each of them mapping either a source CFG to another source CFG, or an IR CFG to another IR CFG. We assume that the code transformations performed by $O_1, \ldots, O_w$ preserve semantics.

We associate each resource counter $r_i$ for $i = 1 \cdots s$, with a non-empty subset $\sigma(r_i)$ of $\{O_1, \ldots, O_w\}$, such that we have $f_i(O(\mathcal{S})) \leq f_i(\mathcal{S})$, for all $O \in \sigma(r_i)$. Hence, $\sigma(r_i)$ is a subset of the optimization strategies among $O_1, \ldots, O_w$ that have the potential to reduce $r_i$. Of course, the intention is that for at least one $O \in \sigma(r_i)$, we have $f_i(O(\mathcal{S})) < f_i(\mathcal{S})$. A reason for not finding such $O$ would be that $\mathcal{S}$ cannot be further optimized w.r.t. $r_i$. We also make a natural idempotence assumption: $f_i(O(O(\mathcal{S}))) = f_i(O(\mathcal{S}))$, for all $O \in \sigma(r_i)$. Similarly, we associate each performance counter $p_i$, for $i = 1 \cdots t$, with a non-empty subset $\sigma(p_i)$ of $\{O_1, \ldots, O_w\}$, such that we have $g_i(O(\mathcal{S})) \geq g_i(\mathcal{S})$ and $g_i(O(O(\mathcal{S}))) = g_i(O(\mathcal{S}))$, for all $O \in \sigma(p_i)$. Hence, $\sigma(p_i)$ is a subset of the optimization strategies among $O_1, \ldots, O_w$ that have the potential to increase $p_i$. The intention is, again, that for at least one $O \in \sigma(p_i)$, we have $g_i(O(\mathcal{S})) > g_i(\mathcal{S})$.

### 3.5 Comprehensive optimization

Let $C_1, \ldots, C_e$ be semi-algebraic systems (that is, conjunctions of polynomial equations and inequalities) with $P_1, \ldots, P_t$, $R_1, \ldots, R_s$, $D_1, \ldots, D_u$, $E_1, \ldots, E_v$ as indeterminate variables. Let $\mathcal{S}_1, \ldots, \mathcal{S}_e$ be fragments of C programs such that the parameters of each of them are among $D_1, \ldots, D_u$, $E_1, \ldots, E_v$.

**Definition 2** The sequence $(C_1, \mathcal{S}_1), \ldots, (C_e, \mathcal{S}_e)$ is a comprehensive optimization of $\mathcal{S}$ w.r.t. if the following conditions hold:

1. **[constraint soundness]** Each system $C_1, \ldots, C_e$ is consistent, that is, admits at least one real solution.

2. **[code soundness]** For all real values $h_1, \ldots, h_t$, $x_1, \ldots, x_s$, $y_1, \ldots, y_u$, $z_1, \ldots, z_v$ of $P_1, \ldots, P_t$, $R_1, \ldots, R_s$, $D_1, \ldots, D_u$, $E_1, \ldots, E_v$ respectively, for all $i \in \{1, \ldots, e\}$ such that $(h_1, \ldots, h_t, x_1, \ldots, x_s, y_1, \ldots, y_u, z_1, \ldots, z_v)$ is a solution of $C_i$, then the code fragment $\mathcal{S}_i$ produces the same output as $\mathcal{S}$ on any data that makes $\mathcal{S}$ execute correctly.

3. **[coverage]** For all real values $y_1, \ldots, y_u, z_1, \ldots, z_v$ of $D_1, \ldots, D_u$, $E_1, \ldots, E_v$, respectively, there exist $i \in \{1, \ldots, e\}$ and real values $h_1, \ldots, h_t$, $x_1, \ldots, x_s$ of $P_1, \ldots, P_t$, $R_1, \ldots, R_s$, such that $(h_1, \ldots, h_t, x_1, \ldots, x_s, y_1, \ldots, y_u, z_1, \ldots, z_v)$ is a solution of $C_i$ and $\mathcal{S}_i$ produces the same output as $\mathcal{S}$ on any data that makes $\mathcal{S}$ execute correctly.

4. **[optimality]** For every $i \in \{1, \ldots, s\}$ (resp. $\{1, \ldots, t\}$), there exists $\ell \in \{1, \ldots, e\}$ such that for all $O \in \sigma(r_i)$ (resp. $\sigma(p_i)$) we have $f_i(O(\mathcal{S}_i)) = f_i(\mathcal{S}_i)$ (resp. $g_i(O(\mathcal{S}_i)) = g_i(\mathcal{S}_i)$).

We summarize Definition 2 in non technical terms. Condition (i) states that each system of constraints is meaningful. Condition (ii) states that as long as the machine, program and data parameters satisfy $C_i$, the code fragment $\mathcal{S}_i$ produces the same output as $\mathcal{S}$ on whichever data that makes $\mathcal{S}$ execute correctly. Condition (iii) states that as long as $\mathcal{S}$ executes correctly on a given set of parameters and data, there
exists a code fragment $\mathcal{S}_i$, for suitable values of the machine parameters, such that $\mathcal{S}_i$ produces the same output as $\mathcal{S}$ on that set of parameters and data. Finally, Condition (iv) states that for each resource counter $r_i$ (performance counter $p_i$), there exists at least one code fragment $\mathcal{S}_i$ for which this counter is optimal in the sense that it cannot be further optimized by the optimization strategies from $\sigma(r_i)$ (resp. $\sigma(p_i)$).

3.6 Data-structures

The algorithm presented in Section 3.7 computes a comprehensive optimization of $\mathcal{S}$ w.r.t. the evaluation functions $f_1, \ldots, f_t, g_1, \ldots, g_t$ and optimization strategies $O_1, \ldots, O_w$. Hereafter, we define the main data-structure used during the course of the algorithm. We associate $\mathcal{S}$ with what we call a quintuple, denoted by $Q(\mathcal{S})$ and defined as follows: $Q(\mathcal{S}) = (G_C(\mathcal{S}), \lambda(\mathcal{S}), \omega(\mathcal{S}), \gamma(\mathcal{S}), C(\mathcal{S}))$, where

1. $\lambda(\mathcal{S})$ is the sequence of the optimization procedures among $O_1, \ldots, O_w$ that have already been applied to the IR of $\mathcal{S}$; hence, $G_C(\mathcal{S})$ together with $\lambda(\mathcal{S})$ defines $G_L(\mathcal{S})$; initially, $\lambda(\mathcal{S})$ is empty,
2. $\omega(\mathcal{S})$ is the sequence of the optimization procedures among $O_1, \ldots, O_w$ that have not been applied so far to either $G_C(\mathcal{S})$ or $G_L(\mathcal{S})$; initially, $\omega(\mathcal{S})$ is $O_1, \ldots, O_w$,
3. $\gamma(\mathcal{S})$ is the sequence of resource and performance counters that remain to be evaluated on $\mathcal{S}$; initially, $\gamma(\mathcal{S})$ is $r_1, \ldots, r_t, p_1, \ldots, p_t$,
4. $C(\mathcal{S})$ is the sequence of the polynomial constraints on $P_1, \ldots, P_t, R_1, \ldots, R_s, D_1, \ldots, D_u, E_1, \ldots, E_v$ that have been computed so far; initially, $C(\mathcal{S})$ is $1 \geq P_1 \geq 0, \ldots, 1 \geq P_t \geq 0, R_1 \geq 0, \ldots, R_s \geq 0, D_1 \geq 0, \ldots, D_u \geq 0, E_1 \geq 0, \ldots, E_v \geq 0$.

We say that the quintuple $Q(\mathcal{S})$ is processed whenever $\gamma(\mathcal{S})$ is empty; otherwise, we say that $Q(\mathcal{S})$ is in-process.

Remark 2 For the above $Q(\mathcal{S})$, each of the sequences $\lambda(\mathcal{S}), \omega(\mathcal{S}), \gamma(\mathcal{S})$ and $C(\mathcal{S})$ is implemented as a stack in Algorithms 1 and 2. Hence, we need to specify how operations on a sequence is performed on the corresponding stack. Let $s_1, s_2, \ldots, s_N$ is a sequence.

1. Popping one element out of this sequence returns $s_1$ and leaves that sequence with $s_2, \ldots, s_N$,
2. Pushing an element $t_1$ on $s_1, s_2, \ldots, s_N$ will update that sequence to $t_1, s_1, s_2, \ldots, s_N$.
3. Pushing a sequence of elements $t_1, t_2, \ldots, t_M$ on $s_1, s_2, \ldots, s_N$ will update that sequence to $t_M, t_1, t_2, t_3, \ldots, s_N$.

3.7 The algorithm

Algorithm 1 is the top-level procedure. If its input is a processed quintuple $Q(\mathcal{S})$, then it returns the pair $(G_C(\mathcal{S}), \lambda(\mathcal{S}))$ (such that, after optimizing $\mathcal{S}$ with the optimization strategies in $\lambda(\mathcal{S})$, one can generate the IR of the optimized $\mathcal{S}$) together with the system of constraints $C(\mathcal{S})$. Otherwise, Algorithm 1 is called recursively on each quintuple returned by Optimize($Q(\mathcal{S})$). The pseudo-code of the Optimize routine is given by Algorithm 2.

We make a few observations about Algorithm 2:

(R1) Observe that at Line (5), a deep copy of the input $Q(\mathcal{S}')$ is made, and this copy is called $Q(\mathcal{S}''')$. This duplication allows the computations to fork. Note that at Line (6), $Q(\mathcal{S}'')$ is modified.

(R2) In this forking process, we call $Q(\mathcal{S}'')$ the accept branch and $Q(\mathcal{S}''')$ the refuse branch. In the former case, the relation $0 \leq v_i \leq R_i$ holds thus implying that enough $R_i$-resources are available for executing the code fragment $\mathcal{S}'$. In the latter case, the relation $R_i < v_i$ holds thus implying that not enough $R_i$-resources are available for executing the code fragment $\mathcal{S}'''$.

(R3) At Lines (18-20), a similar forking process occurs. Here again, we call $Q(\mathcal{S}'')$ the accept branch and $Q(\mathcal{S}''')$ the refuse branch. In the former case, the relation $0 \leq v_i \leq P_i$ implies that the $P_i$-performance counter may have reached its maximum ratio; hence, no optimization strategies are applied to improve this counter. In the latter case, the relation $P_i < v_i \leq 1$ holds thus implying that
the $P_i$-performance counter has not reached its maximum value; hence, optimization strategies are applied to improve this counter if such optimization strategies are available. Observe that if this optimization strategy does make the estimated value of $P_i$ larger then an algebraic contradiction would happen and the branch will be discarded.

(R4) Line (30) in Algorithm 2 requires non-trivial computations with polynomial equations and inequalities. The algorithms can be found in [10] and are implemented in the RegularChains library of MAPLE.

(R5) Each system of algebraic constraints $C$ is updated by adding a polynomial inequality to it at either Lines (6), (7), (19) or (20). This incremental process can be performed by the RealTriangularize algorithm [10] and implemented in the RegularChains library.

(R6) Because of the recursive calls at Lines (16) and (29) several inequalities involving the same variable among $R_1, \ldots, R_s, P_1, \ldots, P_t$ may be added to a given system $C$. As a result, $C$ may become inconsistent. For instance if $R_1 < 10$ and $10 \leq R_1$ are added to the same system $C$. This will happen when an optimization strategy fails to improve the value of a resource or performance counter. Note that inconstancy is automatically detected by the RealTriangularize algorithm.

(a) The decision subtree for resource counter $r_i$

(b) The decision subtree for performance counter $p_i$

Figure 5: The decision subtree for resource or performance counters

We associate the execution of Algorithm 1 applied to $Q(S)$, with a tree denoted by $T(Q(S))$ and where both nodes and edges of $T(Q(S))$ are labelled. We use the same notations as in Algorithm 2. We define $T(Q(S))$ recursively:

(T1) We label the root of $T(Q(S))$ with $Q(S)$.

(T2) If $\gamma(S)$ is empty, then $T(Q(S))$ has no children; otherwise, two cases arise:

(T2.1) If no optimization strategy is to be applied for optimizing the counter $c$, then $T(Q(S))$ has a single subtree, which is that associated with Optimize($Q(S')$) where $Q(S')$ is obtained
from $Q(\mathcal{I})$ by augmenting $C(\mathcal{I})$ either with $0 \leq v_i \leq R_i$ if $c$ is a resource counter or with $0 \leq v_i \leq P_i$ otherwise.

(T2.2) If an optimization strategy is applied, then $\mathcal{T}(Q(\mathcal{I}))$ has two subtrees:

(T2.2.1) The first one is the tree associated with Optimize($Q(\mathcal{I}')$) (where $Q(\mathcal{I}')$ is defined as above) and is connected to its parent node by the accept edge, labelled with either $0 \leq v_i \leq R_i$ or $0 \leq v_i \leq P_i$; see Figure 5.

(T2.2.2) The second one is the tree associated with Optimize($Q(\mathcal{I}'')$) (where $Q(\mathcal{I}'')$ is obtained by applying the optimization strategy to the deep copy of the input quintuple $Q(\mathcal{I})$) and is connected to its parent node by the refuse edge, labelled with either $R_i < v_i$ or $P_i < v_i \leq 1$; see Figure 5.

Observe that every node of $\mathcal{T}(Q(\mathcal{I}))$ is labelled with a quintuple and every edge with a polynomial constraint.

Figure 5 illustrates how Algorithm 2, applied to $Q(\mathcal{I}')$, generates the associated tree $\mathcal{T}(Q(\mathcal{I}))$. The cases for a resource counter and a performance counter are distinguished in the sub-figures (a) and (b), respectively. Observe that, in both cases, the accept edges go south-east, while the refuse edges go south-west.

---

**Algorithm 1: ComprehensiveOptimization**

**Input:** The quintuple $Q(\mathcal{I})$

**Output:** A comprehensive optimization of $\mathcal{I}$ w.r.t. the resource evaluation functions $f_1, \ldots, f_s$, the performance evaluation functions $g_1, \ldots, g_t$, and the optimization strategies $O_1, \ldots, O_w$

if $\mathcal{I}(\mathcal{I})$ is empty then
  return $((C(\mathcal{I}), \lambda(\mathcal{I})), C(\mathcal{I}))$;

The output stack is initially empty;

for each $Q(\mathcal{I}') \in \text{Optimize}(Q(\mathcal{I}))$ do
  Push ComprehensiveOptimization($Q(\mathcal{I}')$) on the output stack;

return the output stack;

---

**Lemma 1** The height of the tree $\mathcal{T}(Q(\mathcal{I}))$ is at most $w(s + t)$. Therefore, Algorithm 1 terminates.

**Proof** Consider a path $\Gamma$ from the root of $\mathcal{T}(Q(\mathcal{I}))$ to any node $N$ of $\mathcal{T}(Q(\mathcal{I}))$. Observe that $\Gamma$ counts at most $w$ refuse edges. Indeed, following a refuse edge decreases by one the number of optimization strategies to be used. Observe also that the length of every sequence of consecutive accept edges is at most $s + t$. Indeed, following an accept edge decreases by one the number of resource and performance counters to be evaluated. Therefore, the number of edges in $\Gamma$ is at most $w(s + t)$. ⊥

**Lemma 2** Let $U := \{U_1, \ldots, U_z\}$ be a subset of $\{O_1, \ldots, O_w\}$. There exists a path from the root of $\mathcal{T}(Q(\mathcal{I}))$ to a leaf of $\mathcal{T}(Q(\mathcal{I}))$ along which the optimization strategies being applied are exactly those of $U$.

**Proof** Let us start at the root of $\mathcal{T}(Q(\mathcal{I}))$ and apply the following procedure:

1. follow the refuse edge if it uses an optimization strategy from $\{U_1, \ldots, U_z\}$,
2. follow the accept edge, otherwise.

This creates a path from the root of $\mathcal{T}(Q(\mathcal{I}))$ to a leaf with the desired property. ⊥

**Definition 3** Let $i \in \{1, \ldots, s\}$ (resp. $\{1, \ldots, t\}$). Let $N$ be a node of $\mathcal{T}(Q(\mathcal{I}))$ and $Q(\mathcal{I}_N)$ be the quintuple labelling this node. We say that $r_i$ (resp. $p_i$) is optimal at $N$ w.r.t. the evaluation function $f_i$ (resp. $g_i$) and the subset $\sigma(r_i)$ (resp. $\sigma(p_i)$) of the optimization strategies $O_1, \ldots, O_w$, whenever for all $O \in \sigma(r_i)$ (resp. $\sigma(p_i)$) we have $f_i(O(\mathcal{I}_N)) = f_i(\mathcal{I}_N)$ (resp. $g_i(O(\mathcal{I}_N)) = g_i(\mathcal{I}_N)$).
Algorithm 2: Optimize

**Input:** A quintuple \( Q(\mathcal{F}) \)

**Output:** A stack of quintuples

Initialize an empty stack, called \( \text{result} \);

Take out from \( \gamma(\mathcal{F}') \) the next resource or performance counter to be evaluated, say \( c \);

Evaluate \( c \) on \( \mathcal{F}' \) (using the appropriate functions among \( f_1, \ldots, f_s, g_1, \ldots, g_t \)) thus obtaining a value \( v_i \), which can be either a numerical value, a polynomial in \( \mathbb{Q}[D_1, \ldots, D_u, E_1, \ldots, E_v] \) or a rational function where its numerator and denominator are in \( \mathbb{Q}[D_1, \ldots, D_u, E_1, \ldots, E_v, R_1, \ldots, R_w] \);

if \( c \) is a resource counter \( r_i \) then

Make a deep copy \( Q(\mathcal{F}'') \) of \( Q(\mathcal{F}') \), since we are going to split the computation into two branches:

\[ R_i < v_i \quad \text{and} \quad 0 \leq v_i \leq R_i; \]

Add the constraint \( 0 \leq v_i \leq R_i \) to \( C(\mathcal{F}') \) and push \( Q(\mathcal{F}') \) onto \( \text{result} \);

Add the constraint \( R_i < v_i \) to \( C(\mathcal{F}') \) and search \( \omega(\mathcal{F}'') \) for an optimization strategy of \( \sigma(r_i) \);

if no such optimization strategy exists then

\[ \text{return} \ \text{result}; \]

else

Apply such an optimization strategy to \( Q(\mathcal{F}'') \) yielding \( Q(\mathcal{F}'')' \);

Remove this optimization strategy from \( \omega(\mathcal{F}'') \);

if this optimization strategy is applied to the IR of \( \mathcal{F}'' \) then

\[ \text{Push } r_1, \ldots, r_{i-1}, r_i \text{ onto } \gamma(\mathcal{F}''); \]

Make a recursive call to Optimize on \( Q(\mathcal{F}'') \) and push the returned quintuples onto \( \text{result} \);

if \( c \) is a performance counter \( p_i \) then

Make a deep copy \( Q(\mathcal{F}'') \) of \( Q(\mathcal{F}') \), since we are going to split the computation into two branches:

\[ 0 \leq v_i < P_i \quad \text{and} \quad P_i < v_i \leq 1; \]

Add the constraint \( 0 \leq v_i \leq P_i \) to \( C(\mathcal{F}') \) and push \( Q(\mathcal{F}') \) onto \( \text{result} \);

Add the constraint \( P_i < v_i \leq 1 \) to \( C(\mathcal{F}') \) and search \( \omega(\mathcal{F}'') \) for an optimization strategy of \( \sigma(p_i) \);

if no such optimization strategy exists then

\[ \text{return} \ \text{result}; \]

else

Apply such an optimization strategy to \( Q(\mathcal{F}'') \) yielding \( Q(\mathcal{F}'')' \);

Remove this optimization strategy from \( \omega(\mathcal{F}'') \);

if this optimization strategy is applied to the IR of \( \mathcal{F}'' \) then

\[ \text{Push } r_1, \ldots, r_s, p_i \text{ onto } \gamma(\mathcal{F}''); \]

Make a recursive call to Optimize on \( Q(\mathcal{F}'') \) and push the returned quintuples onto \( \text{result} \);

Remove from \( \text{result} \) any quintuple with an inconsistent system of constraints;

\[ \text{return} \ \text{result}; \]

---

**Lemma 3** Let \( i \in \{1, \ldots, s\} \) (resp. \( \{1, \ldots, t\} \)). There exists at least one leaf \( L \) of \( \mathcal{T}(Q(\mathcal{F})) \) such that \( r_i \) (resp. \( p_i \)) is optimal at \( L \) w.r.t. the evaluation function \( f_i \) (resp. \( g_i \)) and the subset \( \sigma(r_i) \) (resp. \( \sigma(p_i) \)) of the optimization strategies \( O_1, \ldots, O_w \).

**Proof** Apply Lemma 2 with \( U = \sigma(r_i) \) (resp. \( U = \sigma(p_i) \)).

---

**Lemma 4** Algorithm 4 satisfies its output specifications.

**Proof** From Lemma 1, we know that Algorithm 1 terminates. So let \( (C_1, \mathcal{F}_1), \ldots, (C_e, \mathcal{F}_e) \) be its output. We shall prove \( (C_1, \mathcal{F}_1), \ldots, (C_e, \mathcal{F}_e) \) satisfies the conditions (i) to (iv) of Definition 2.

Condition (i) is satisfied by the properties of the RealTriangularize algorithm. Condition (ii) follows clearly from the assumption that the code transformations performed by \( O_1, \ldots, O_w \) preserve semantics. Observe that each time a polynomial inequality is added to a system of constraints, the negation
of this inequality is also to the same system in another branch of the computations. By using a simple induction on \(s + t\), we deduce that Condition \((iii)\) is satisfied. Finally, we prove Condition \((iv)\) by using Lemma 5.

4 Comprehensive Translation

Given a high-level model for accelerator programming (like OPENCL [23], OPENMP, OPENACC or MetaFork [11]), we consider the problem of translating a program written for such a high-level model into a programming model for GPGPU devices, such as CUDA. We assume that the numerical values of some, or all, of the hardware characteristics of the targeted GPGPU device are unknown. Hence, these quantities are treated as symbols. Similarly, we would like that some, or all, of the program parameters remain symbols in the generated code.

In our implementation, we focus on one high-level model for accelerator programming, namely MetaFork. However, we believe that an adaptation to another high-level model for accelerator programming would not be difficult. One supporting reason for that claim is the fact that automatic code translation between the MetaFork and OpenMP languages can already be done within the MetaFork compilation framework, see [11].

We consider as input a meta_schedule statement \(\mathcal{M}\) and its surrounding MetaFork program \(\mathcal{P}\). In our implementation, we assume that, apart from the meta_schedule statement \(\mathcal{M}\), the rest of the program \(\mathcal{P}\) is serial C code. Now, applying the comprehensive optimization algorithm (described in Section 3) on the meta_schedule statement \(\mathcal{M}\) (with prescribed resource evaluation functions, performance evaluation functions and optimization strategies) we obtain a sequence of processed quintuples of meta_schedule statements \(\mathcal{Q}_1(\mathcal{M}), \mathcal{Q}_2(\mathcal{M}), \ldots, \mathcal{Q}_\ell(\mathcal{M})\), which forms a comprehensive optimization in the sense of Definition 2.

If, as mentioned in the introduction, PTX is used as intermediate representation (IR) then, for each \(i = 1, \ldots, \ell\), under the constraints defined by the polynomial system associated with \(\mathcal{Q}_i(\mathcal{M})\), the IR code associated with \(\mathcal{Q}_i(\mathcal{M})\) is the translation in assembly language of a CUDA counterpart of \(\mathcal{M}\). In our implementation, we also translate to CUDA source code the MetaFork code in each \(\mathcal{Q}_i(\mathcal{M})\), since this is easier to read for a human being.

5 Experimentation

We report on a preliminary implementation of Algorithm 1 dedicated to the optimization of meta_schedule statements in view of generating parametric CUDA kernels. Two hardware resource counters are considered: register usage per thread, and local/shared memory allocated per thread-block. No performance counters are specified, however, by design, the algorithm tries to minimize the usage of hardware resources. Four optimization strategies are used: (i) reducing register pressure; (ii) controlling thread granularity; (iii) common sub-expression elimination (CSE), and (iv) caching data in local/shared memory. The first one applies to the IR CFG and uses LLVM; (ii) and (iii) are performed in MAPLE on the source CFG while (iv) combines PET [26] and the RegularChains library in MAPLE. Moreover, for (i) and (iii), 3 and 2 levels of optimization are used, respectively. Figure 6 gives an overview of the software tools that are involved in our implementation.

The test examples of our CASCON paper [9] have been extensively tested with that implementation. In the interest of space, we have selected two representative examples. For both of them, again in the interest of space, we present the optimized MetaFork code, instead of the optimized IR code (or the CUDA code generated from MetaFork).

\[^5\] In the MetaFork language, the keyword cache is used to indicate that every thread accessing a specified array \(a\) must copy in local/shared memory the data it accesses in \(a\).
Figure 6: The software tools involved in our implementation

For each system of constraints, we indicate which decisions were made by the algorithm to reach that case. To this end, we use the following abbreviations: (1), (2), (3a), (3b), (4a), (4b) respectively stand for “No register pressure optimization”, “CSE is Applied” “thread granularity not-reduced”, “reduced thread granularity”, “Use local/shared memory”, “Do not use local/shared memory”.

For both test-examples, we give speedup factors (against an optimized serial C code) obtained with the most efficient of our generated CUDA kernels. All CUDA experimental results are collected on an NVIDIA Tesla M2050.

5.1 1D Jacobi

Both source and optimized METAfork programs are shown on Figure 7. Table 2 shows the speedup factors for the first case of optimized METAfork programs in Figure 7.

| Thread-block \ Granularity | 2  | 4  | 8  |
|----------------------------|----|----|----|
| 16                         | 3.340 | 4.357 | 4.975 |
| 32                         | 4.785 | 5.252 | 5.206 |
| 64                         | 5.927 | 6.264 | 6.412 |
| 128                        | **10.400** | 8.952 | 5.793 |
| 256                        | 6.859 | 6.246 |    |

Table 2: Speedup factors of 1D Jacobi for time iteration 4 and input vector of length $2^{15} + 2$

5.2 Matrix transposition

Three cases of optimized METAfork programs are shown on Figure 8. Table 3 shows the speedup factors for the first case of optimized METAfork programs in Figure 8.
6 Concluding Remarks

We have shown how, from an annotated C/C++ program, parametric CUDA kernels could be optimized. These optimized parametric CUDA kernels are organized in the form of a case discussion, where cases depend on the values of machine parameters (e.g. hardware resource limits) and program parameters (e.g. source code

```
int T, N, s, B,
int dim = (N-2)/(s*B);
int a[2*N];
for (int t = 0; t<T; ++t)
    meta_schedule {
        meta_for (int i = 0; i<dim; i++)
        meta_for (int j = 0; j<B; j++)
        for (int k = 0; k<s; ++k) {
            int p = i*s*B+k*B+j;
            int p1 = p + 1;
            int p2 = p + 2;
            int np = N + p;
            int np1 = N + p + 1;
            int np2 = N + p + 2;
            if (t % 2)
                a[p1] = (a[np]+a[np1]+a[np2])/3;
            else
                a[np1] = (a[p]+a[p1]+a[p2])/3;
        }
    }
```

First case

```
\begin{align*}
2sB + 2 &\leq ZB \\
ZB &< 2sB + 2 \\
9 &\leq RB
\end{align*}
```

for (int t = 0; t<T; ++t)
    meta_schedule cache(a) {
        meta_for (int i = 0; i<dim; i++)
        meta_for (int j = 0; j<B; j++)
        for (int k = 0; k<s; ++k) {
            int p = j+(i+s*k)*B;
            int t16 = p+1;
            int t15 = p+2;
            int p1 = t16;
            int p2 = t15;
            int np = N+p;
            int np1 = N+t16;
            int np2 = N+t15;
            if (t % 2)
                a[p1] = (a[np]+a[np1]+a[np2])/3;
            else
                a[np1] = (a[p]+a[p1]+a[p2])/3;
        }
    }
```

Second case

```
\begin{align*}
ZB &< 2sB + 2 \\
9 &\leq RB
\end{align*}
```

for (int t = 0; t<T; ++t)
    meta_schedule cache(a) {
        meta_for (int i = 0; i<dim; i++)
        meta_for (int j = 0; j<B; j++)
        for (int k = 0; k<s; ++k) {
            int p = i*B+j;
            int t20 = p+1;
            int t19 = p+2;
            int p1 = t20;
            int p2 = t19;
            int np = N+p;
            int np1 = N+t20;
            int np2 = N+t19;
            if (t % 2)
                a[p1] = (a[np]+a[np1]+a[np2])/3;
            else
                a[np1] = (a[p]+a[p1]+a[p2])/3;
        }
    }
```

Third case

```
\begin{align*}
ZB &< 2sB + 2 \\
9 &\leq RB
\end{align*}
```

for (int t = 0; t<T; ++t)
    meta_schedule {
        meta_for (int i = 0; i<dim; i++)
        meta_for (int j = 0; j<B; j++)
        for (int k = 0; k<s; ++k) {
            int p = j+(i+s*k)*B;
            int t16 = p+1;
            int t15 = p+2;
            int p1 = t16;
            int p2 = t15;
            int np = N+p;
            int np1 = N+t16;
            int np2 = N+t15;
            if (t % 2)
                a[p1] = (a[np]+a[np1]+a[np2])/3;
            else
                a[np1] = (a[p]+a[p1]+a[p2])/3;
        }
    }
```

Figure 7: Three optimized METAORK programs for 1D Jacobi
```c
meta_schedule cache(a, c) {
    meta_for (int v0 = 0; v0<dim0; v0++)
    meta_for (int v1 = 0; v1<dim1; v1++)
    meta_for (int u0 = 0; u0<B0; u0++)
    meta_for (int u1 = 0; u1<B1; u1++)
        for (int k = 0; k < s; ++k) {
            int i = v0*B0+u0;
            int j = (v1*s+k)*B1+u1;
            c[i*N+j] = a[i][j];
        }
}
```

Second case
```c
meta_schedule cache(a, c) {
    meta_for (int v0 = 0; v0<dim0; v0++)
    meta_for (int v1 = 0; v1<dim1; v1++)
    meta_for (int u0 = 0; u0<B0; u0++)
    meta_for (int u1 = 0; u1<B1; u1++)
        int i = v0*B0+u0;
        int j = (v1*s+k)*B1+u1;
        c[i*N+j] = a[i][j];
    }
}
```

Third case
```c
meta_schedule {
    meta_for (int v0 = 0; v0<dim0; v0++)
    meta_for (int v1 = 0; v1<dim1; v1++)
    meta_for (int u0 = 0; u0<B0; u0++)
    meta_for (int u1 = 0; u1<B1; u1++)
        int i = v0*B0+u0;
        int j = v1*B1+u1;
        c[i*N+j] = a[i][j];
    }
}
```

Figure 8: Three optimized METAFORK programs for matrix transposition

| Thread-block \ Granularity | 2     | 4     | 8    |
|----------------------------|-------|-------|------|
| (4, 32)                    | 103.281 | 96.284 | 75.211 |
| (16, 32)                   | 111.971 | 90.625 | 85.422 |
| (32, 32)                   | 78.476  | 68.894 | 48.822 |
| (128, 128)                 | 45.084  | 46.425 | 32.824 |

Table 3: Speedup factors of matrix transposition for input matrix of order $2^{14}$

dimension sizes of thread-blocks).

The proposed approach extend previous works, in particular PPCG [25] and CUDA-CH1LL [16], and combine them with techniques from computer algebra. Indeed, handling systems of non-linear polynomial equations and inequalities is required in the context of parametric CUDA kernels.

Our preliminary implementation uses LLVM, MAPLE and PPCG; it successfully processes a variety of standard test-examples. In particular, the computer algebra portion of the computations is not a bottleneck.

Acknowledgments

The authors would like to thank the IBM Toronto labs and NSERC of Canada for supporting their work.

References

[1] User guide for NVPTX. The LLVM Compiler Infrastructure. [http://llvm.org/docs/NVPTXUsage.html#introduction](http://llvm.org/docs/NVPTXUsage.html#introduction)

[2] CUDA runtime API: v7.5. NVIDIA Corporation, 2015. [http://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf](http://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf)
[3] The OpenACC application programming interface. OpenACC-Standard.org, 2015.
[4] OpenMP application program interface version 4.5. OpenMP Architecture Review Board, 2015. [http://www.openmp.org/mp-documents/openmp-4.5.pdf]
[5] Parallel thread execution ISA : v4.3. NVIDIA Corporation, 2015. [http://docs.nvidia.com/cuda/pdf/ptx_isa_4.3.pdf]
[6] Ravi Sethi Aho, Alfred V. and Jeffrey D. Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley, Reading, Massachusetts, 2006.
[7] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In Proceedings of CC ’10/ETAPS ’10, pages 244–263, Berlin, Heidelberg, 2010. Springer-Verlag.
[8] Carlo Bertolli, Samuel F. Antao, Alexandre E. Eichenberger, Kevin O’Brien, Zehra Sura, Arpith C. Jacob, Tong Chen, and Olivier Sallenave. Coordinating GPU threads for OpenMP 4.0 in LLVM. In Proceedings of LLVM-HPC ’14, pages 12–21. IEEE Press, 2014.
[9] Changbo Chen, Xiaohui Chen, Abdul-Kader Keita, Marc Moreno Maza, and Ning Xie. MetaFork: A compilation framework for concurrency models targeting hardware accelerators and its application to the generation of parametric CUDA kernels. In Proceedings of CASCON 2015, pages 70–79, 2015.
[10] Changbo Chen, James H. Davenport, John P. May, Marc Moreno Maza, Bican Xia, and Rong Xiao. Triangular decomposition of semi-algebraic systems. J. Symb. Comput., 49:3–26, 2013.
[11] Xiaohui Chen, Marc Moreno Maza, Sushek Shekar, and Priya Unnikrishnan. MetaFork: A framework for concurrency platforms targeting multicore processors. In Proceeding of IWOMP 2014, pages 30–44, 2014.
[12] Leonardo Dagum and Ramesh Menon. OpenMP: An industry standard API for shared-memory programming. Computational Science & Engineering, IEEE, 5(1):46–55, 1998.
[13] Matteo Frigo and Steven G. Johnson. FFTW: an adaptive software architecture for the FFT. In Proceedings of ICASSP, pages 1381–1384. IEEE, 1998.
[14] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee AyalaRohatgi, and John Cavazos. Auto-tuning a high-level language targeted to GPU codes. In Innovative Parallel Computing. IEEE, 2012.
[15] Tianyi David Han and Tarek S. Abdelrahman. hiCUDA: A high-level directive-based language for GPU programming. In Proceedings of GPGPU-2, pages 52–61. ACM, 2009.
[16] Malik Khan, Protonu Basu, Gabe Rudy, Mary Hall, Chun Chen, and Jacqueline Chame. A script-based autotuning compiler system to generate high-performance CUDA code. ACM Trans. Archit. Code Optim., 9(4):31:1–31:25, January 2013.
[17] Thomas Kistler and Michael Franz. Continuous program optimization: A case study. ACM Trans. on Programming Languages and Systems, 25(4):500–548, 2003.
[18] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of CGO ’04, pages 75–, IEEE Computer Society, 2004.
[19] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. Queue, 6(2):40–53, 2008.
[20] Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, and Robert W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. IJHPCA, 18(1), 2004.
[21] Helge Rhodin. A PTX code generator for LLVM. Master’s thesis, Saarland University, 2010.
[22] Chenchen Song, Lee-Ping Wang, and Todd J Martinez. Automated code engine for graphical processing units: Application to the effective core potential integrals and gradients. Journal of chemical theory and computation, 2015.
[23] 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.
[24] Xiaonian Tian, Rengan Xu, Yonghong Yan, Zhifeng Yun, Sunita Chandrasekaran, and Barbara M. Chapman. Compiling a high-level directive-based programming model for GPGPUs. In Languages and Compilers for Parallel Computing - 26th Int. Work. Springer, 2013.
[25] 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.
[26] Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool. In Second International Workshop on Polyhedral Compilation Techniques (IMPACT 2012), Paris, France, 2012.
[27] R. Clinton Whaley and Jack Dongarra. Automatically tuned linear algebra software. In PPSC, 1999.