Comprehensive Optimization of Parametric Kernels for Graphics Processing Units

Xiaohui Chen\(^1\), Marc Moreno Maza\(^2\), Jeeva Paudel\(^3\), Ning Xie\(^4\)

\(^1\) AMD, Markham, Ontario, Canada
\(^2\) U. Western Ontario, London, Ontario, Canada moreno@csd.uwo.ca
\(^3\) IBM Canada Ltd, Markham, Ontario, Canada
\(^4\) Huawei Technologies Canada, Markham, Ontario, Canada

Overview

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 [17, 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 [10, 4] and OPENACC [21, 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 [22], C-TO-CUDA [6], HiCUDA [13], CUDA-CHILL [14]) 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.

To deal with these challenges in translating annotated C/C++ programs to CUDA, we propose in [8] 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.

A proof-of-concept implementation, presented in [8] and publicly available[1] uses another high-level model for accelerator programming, called METAORK, that we introduced in [9]. 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.

In this work, our goal is to enhance the framework initiated in [8] 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 is the natural framework for manipulating such systems of constraints; our RegularChains library[2] 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 [12, 14], dynamic instrumentation [15] or both [20]. Rephrasing [14], “those techniques explore empirically different data placement and thread/block mapping strategies, along with

[1]www.metafork.org
[2]This library, shipped with the commercialized computer algebra system MAPLE, is freely available at www.regularchains.org
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 [23], FFTW [11], and SPIRAL [18], 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.

We conclude this extended abstract with an example illustrating 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 METAFORK language. The body of the 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\(^3\) or alternatively, to PTX\(^4\) 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.

\[
C_1 : \begin{cases} B_0 \times B_1 \leq T \\ 14 \leq R \end{cases} \quad C_2 : \begin{cases} B_0 \times B_1 \leq T \\ 10 \leq R < 14 \end{cases}
\]

\[
\text{__global__ void } K1(int *a, int *b, int *c, int N, int B0, int B1) \{
\begin{align*}
\text{int } i &= \text{ blockIdx.y } \times B0 + \text{ threadIdx.y}; \\
\text{int } j &= \text{ blockIdx.x } \times B1 + \text{ threadIdx.x}; \\
\text{if } (i < N \&\& j < N/2) 
\end{align*}

\begin{align*}
a[i+N*j] &= b[i+N*j] + c[i+N*j]; \\
a[i+N*j+N/2] &= b[i+N*j+N/2] + c[i+N*j+N/2]; \\
\end{align*}
\}
\]

\[
\text{dim3 dimBlock(B1, B0);} \\
\text{dim3 dimGrid(N/B1, N/B0);} \\
\text{K1 <<<dimGrid, dimBlock>>> (a, b, c, N, B0, B1);} \\
\]

\[
\text{__global__ void } K2(int *a, int *b, int *c, int N, int B0, int B1) \{
\begin{align*}
\text{int } i &= \text{ blockIdx.y } \times B0 + \text{ threadIdx.y}; \\
\text{int } j &= \text{ blockIdx.x } \times B1 + \text{ threadIdx.x}; \\
\text{if } (i < N \&\& j < N) 
\end{align*}

\begin{align*}
a[i+N*j] &= b[i+N*j] + c[i+N*j]; \\
a[i+N*j+N/2] &= b[i+N*j+N/2] + c[i+N*j+N/2]; \\
\end{align*}
\}
\]

\[
\text{dim3 dimBlock(B1, B0);} \\
\text{dim3 dimGrid(N/B1, N/B0);} \\
\text{K2 <<<dimGrid, dimBlock>>> (a, b, c, N, B0, B1);} \\
\]

Figure 2: A comprehensive parametric CUDA kernel for matrix addition.

While this was a toy-example, advanced test cases can be found in Chapter 7 of the PhD thesis of the first author at

[http://ir.lib.uwo.ca/etd/4429](http://ir.lib.uwo.ca/etd/4429)

---

\(^3\) Quoting Wikipedia: “The LLVM compiler infrastructure project (formerly Low Level Virtual Machine \([16,7]\)) is a framework for developing compiler front ends and back ends”.

\(^4\) The Parallel Thread Execution (PTX) \([5]\) 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 \([1]\), following the work of \([19]\).
Acknowledgments

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

References


