Run-time Support for Parametric Kernels Targeting Graphics Processing Units

Masoud Ataei, Xiaohui Chen, Marc Moreno Maza, Jeeva Paudel, Ettore Tiotto, Lin-Xiao Wang, Ning Xie, Haoze Yuan

Parametric Kernels

High-level models for accelerator programming, like OpenMP and OpenACC, have become an important research direction. With these models, programmers only need to annotate their C/C++ code to indicate which portion of code is to be executed on the device (typically a GPU) and how data is mapped between host and device. 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. Existing software packages 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 may limit *code portability* as well as opportunities for *code optimization*.

```
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 1: MetaFork matrix multiplication with 3 program parameters.

```
__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[B_0][B_0];
 __shared__ int shared_b[B_0][B_1];
 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) * (ub1) + t1] =
             b[(t0 + B0 * c2) * n +
             (private_q + (c5) * (ub1))];
       __syncthreads();
       for (int c6 = 0; c6 < B0; c6 += 1)
         for (int c5 = 0; 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 2: generated CUDA kernel with the same program parameters.

To deal with these challenges in translating annotated C/C++ programs to CUDA, we propose in a CASCON 2015 paper 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 the same paper and publicly at `www.metafork.org`, uses another high-level model for accelerator programming, called MetaFork, that we introduced in an IWOMP 2014 paper. The experimentation shows that the generation of parametric CUDA kernels can lead to significant performance improvement with respect to 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.

Figure 1 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 `meta_for` nested loops) computes `s` coefficients of the product matrix. The blocks in `a`, `b`, `c` have format $\texttt{B0} \times \texttt{B0}$, $\texttt{B0} \times (\texttt{ub1} \cdot \texttt{s})$, $\texttt{B0} \times (\texttt{ub1} \cdot \texttt{s})$. Hence, `s`, `B0` and `ub1` are program parameters. Figure 2 shows a CUDA kernel code generated from the MetaFork code of Figure 1. Observe that `kernel0` takes the *program parameters* `B0` and `ub1` as arguments, whereas non-parametric CUDA kernels only take data parameters (here a, b, c, n).

Comprehensive Optimization

In broad terms, a *comprehensive optimization of parametric* CUDA *program* is a decision tree where: each internal node is a Boolean condition on the machine and program parameters, and, each leaf is a CUDA program $\mathcal{P}$, 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 $\mathcal{P}$. Both source and optimized MetaFork programs for the 1D Jacobi stencil are shown below.

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

$$\begin{cases} 2sB+2 \le Z_B \\ 9 \le R_B \end{cases}$$

(1) (4a) (3a) (2) (2)

```
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

$$\begin{cases} 2B+2 \le Z_B \\ Z_B < 2sB+2 \\ 9 \le R_B \end{cases}$$

(1) (3b) (4a) (3a) (2) (2)

```
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++) {
     int p = i*B+j;
     int t20 = p+1;
     int t19 = p+2;
     int p1 = t20;
     int p2 = t19;
     int np = N+p;
     int np2 = N+t19;
     int np1 = N+t20;
     if (t % 2)
      a[p1] = (a[np]+
        a[np1]+a[np2])/3;
     else
      a[np1] = (a[p]+
        a[p1]+a[p2])/3;
    }
}
```

Third case

$$\begin{cases} Z_B < 2B+2 \\ 9 \le R_B \end{cases}$$

(1) (3b) (2) (2) (4b)

```
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++) {
     int p = j+i*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;
    }
}
```

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 a 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 BPAS library (see the related poster) is meant to provide the appropriate algorithmic tools for that task. An algorithm for such comprehensive optimizations and a proof-of-concept implementation are presented in `https://arxiv.org/abs/1801.04348`.

Run-time Support

In the PhD theses of Xiaohui Chen and Ning Xie, the determination of optimal parameter values of our parametric GPU kernels were done by an exhaustive search. In a new prototype, presented at CASCON 2017, optimal parameter values are computed by solving a mathematical optimization problem at run-time. Experimentally, this strategy produces parameter values instantaneously and shows that parametric GPU kernels not only improve performance significantly but also satisfy the exigence of real time computation. The table shows that this approach recovers the results of Xiaohui Chen and Ning Xie. We sketch how this run-time parameter determination. We start by specifying what is computed *off-line*, that is, at compile time.

▷ Off-line phase ◁

Step 1: Optimization (with case discussion) of parametric kernels (with machine and program parameters). Each case in the discussion consists of

- a system S semi-algebraic constraints on the parameters
- an objective function $\hat{T}$
- an optimized kernel in the form of a pair (MetaFork, LLVM IR) of equivalent codes; of course LLVM IR should be replaced by PTX (planed for 2018).

Step 2: Each pair $(S, \hat{T})$ is turned into a highly-optimized C program (SLP techniques).

Step 3: Each MetaFork code is converted into parametric CUDA code, then compiled to a binary B.

▷ At run-time ◁

Step 4: Each pair $(S, \hat{T})$ is specialized at the machine parameters of the targeted device, say to $(S_0, \hat{T}_0)$

Step 5: Inconsistent S_0's are discarded and for the remaining pairs $(S_0, \hat{T}_0)$, the program parameters are determined using optimization techniques (e.g. ILP executed on GPU).

Step 6: B is executed with program and machine parameters passed as arguments.

Example	N	program parameters
Reversal	16384	$s = 1, B = 512$
MVM	2048	$s = 4, B = 512$
MMM	1024	$s = 8, B_0 = 16, B = 8$
Transpose	2048	$s = 1, B_0 = 16, B_1 = 32$
Jacobi	32768	$s = 1, B = 32$
GaussElim	1024	$s = 8, B = 32$

The parameter s gives how many coefficients each thread computes while B, B_0, B_1 are thread-block dimension sizes.