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

Changbo Chen[1] Xiaohui Chen[2,3] Abdoul-Kader Keita[4]
Marc Moreno Maza[1,2,3]
Ning Xie[2]

[1]CIGIT, Chinese Academy of Sciences, China
[2]University of Western Ontario, Canada
[3]IBM Center for Adavnced Studies (CAS Research), Canada
[3]IBM Canada Laboratory

November 2, 2015

Plan

Background

Automatic parallelization

- Most of our computers rely on parallel processors (multi-cores, GPUs), but most of the programs that we have or write are serial ones.
- Writing efficient parallel programs is very hard, hence automatic generation of parallel code from serial one is a dramatically needed:
- In general, automatic generation of parallel code is even harder, but makes sense for kernels in scientific computing (dense linear and polynomial algebra, stencil computations).

Background

Automatic parallelization

- Most of our computers rely on parallel processors (multi-cores, GPUs), but most of the programs that we have or write are serial ones.
- Writing efficient parallel programs is very hard, hence automatic generation of parallel code from serial one is a dramatically needed:
- In general, automatic generation of parallel code is even harder, but makes sense for kernels in scientific computing (dense linear and polynomial algebra, stencil computations).

From C to CUDA

- In automatic generation of GPU code from C code, it is desirable that the generated GPU code depends on parameters (thread-block size, memory sizez, etc.) so as to improve portability and efficiency.
- Standard techniques (like the polyhedron model) rely on solving systems of linear equations and inequalities.
- However, parametric programs (say, in CUDA) introduce non linear expressions requiring polynomial system solvers, in particular QE tools.

Our work

1. We illustrate the need of CUDA programs (more precisely, kernels) that depend on program parameters and machine parameters.
2. We show that techniques from symbolic computation can effectively handle the necessary algebraic computations (i.e. quantifier elimination).
3. We report on a preliminary implementation of a generator of *parametric* CUDA kernels from input MetaFork programs.
4. Experimental results show the benefits of generating parametric CUDA kernels.
5. Our work relies on two of our other projects: the `RegularChains` library in Maple and the MetaFork compilation framework.

A complete example: Jacobi

```
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];
}
```

Original C code.

```
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 ];
       }
     }
  }
}
```

METAFORK code obtained via quantifier elimination.

In generating CUDA code from C, we use METAFORK as an intermediate language.

A complete example: Jacobi

```
int *dev_a;
int *dev_b;

cudaCheckReturn(cudaMalloc((void **) &dev_a, (N) * sizeof(int)));
cudaCheckReturn(cudaMalloc((void **) &dev_b, (N) * sizeof(int)));

#define floord(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
{
 if (N >= 1) {
   cudaCheckReturn(cudaMemcpy(dev_a, a, (N) * sizeof(int),
                              cudaMemcpyHostToDevice));
   cudaCheckReturn(cudaMemcpy(dev_b, b, (N) * sizeof(int),
                              cudaMemcpyHostToDevice));
 }
 for (int c0 = 0; c0 < T; c0 += 1) {
   {
     dim3 k0_dimBlock(B);
     dim3 k0_dimGrid(ub_v <= 32767 ? ub_v : 32768);
     kernel0 <<<k0_dimGrid, k0_dimBlock, (B+2)*sizeof(int)>>>
                        (dev_a, dev_b, N, T, ub_v, B, c0);
     cudaCheckKernel();
   }
   {
     dim3 k1_dimBlock(B);
     dim3 k1_dimGrid(ub_v <= 32767 ? ub_v : 32768);
     kernel1 <<<k1_dimGrid, k1_dimBlock, (B)*sizeof(int)>>>
                       (dev_a, dev_b, N, T, ub_v, B, c0);
     cudaCheckKernel();
   }
 }
 if (N >= 1) {
   cudaCheckReturn(cudaMemcpy(a, dev_a, (N) * sizeof(int),
                              cudaMemcpyDeviceToHost));
   cudaCheckReturn(cudaMemcpy(b, dev_b, (N) * sizeof(int),
                              cudaMemcpyDeviceToHost));
 }
}
cudaCheckReturn(cudaFree(dev_a));
cudaCheckReturn(cudaFree(dev_b));
                  }
```

Generated CUDA Host code.

```
#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();

       private_p = ((((c1) * (B)) + (t0)) + 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();
    }
}
```

CUDA kernel corresponding to the first loop nest.

Plan

CUDA programs

CUDA kernels

- A CUDA kernel is an SIMT (single instruction, multiple data) code executed by all threads in a block and all blocks in a grid
- The format (i.e. number of dimensions and their sizes) of the thread-blocks and the grid are specified when the kernel is launched.
- Formats are part of the definition of the CUDA program and we call them program parameters.

```
__global__ void kernel0(int *a, int *b, int N, int T, int c0)
{
    int b0 = blockIdx.x;
    int t0 = threadIdx.x;

    #define floord(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
    for (int c1 = 32 * b0; c1 < N - 1; c1 += 1048576)
      if (N >= t0 + c1 + 2 && t0 + c1 >= 1)
        b[t0 + c1] = (((a[t0 + c1 - 1] + a[t0 + c1]) + a[t0 + c1 + 1]) / 3);
}
```

Above is a (non-parametric) version of the 1D Jacobi kernel shown before.
In the host code, grid and thread-blocks are 1D with 32 threads per block.

Performance of CUDA programs

Execution of a kernel

- All threads of a thread-block are executed logically in parallel by a streaming multi-processor (SM).
- In fact, an SM decomposes a thread-block into warps of (typically) 32 threads which are physically executed in parallel by the SM.
- One SM may execute more than one thread-block concurrently. Of course, there are hhardware limits on the number of thread-blocks, warps and threads that can be active on an SM.

Performance of CUDA programs

Execution of a kernel

- All threads of a thread-block are executed logically in parallel by a streaming multi-processor (SM).
- In fact, an SM decomposes a thread-block into warps of (typically) 32 threads which are physically executed in parallel by the SM.
- One SM may execute more than one thread-block concurrently. Of course, there are hhardware limits on the number of thread-blocks, warps and threads that can be active on an SM.

Several performance measures

- Total time spent in transfering data between the SMs and the main (i.e. global) memory of the GPU device
- Hardware occupancy, that is, the ratio of the number of active warps on an SM to the maximum number of warps
- Arithmetic intensity, that is, the number of arithmetic operations per second (to be compared to the hardware limit).
- Effective memory bandwidth, that is, the number of bytes read or written from the global memory per second (to be coma-pared to the hardware limit).

Program & hardware parameters and their relation to performance

Program & machine parameters

- GPU devices depend on hardware parameters: size of the memory of an SM, maximum number of registers per thread, memory latencies, etc.
- CUDA programs depend on program parameters (like formats) that specify how the work is distributed among threads and thread-blocks.
- Determining appropriate program parameters is essential for code organization and, even more, for code performance.
- Experimentation shows that program parameters depend on data sizes and machine parameters, which may not be known at compile time.

Program & hardware parameters and their relation to performance

Program & machine parameters

- GPU devices depend on hardware parameters: size of the memory of an SM, maximum number of registers per thread, memory latencies, etc.
- CUDA programs depend on program parameters (like formats) that specify how the work is distributed among threads and thread-blocks.
- Determining appropriate program parameters is essential for code organization and, even more, for code performance.
- Experimentation shows that program parameters depend on data sizes and machine parameters, which may not be known at compile time.

Code optimization techniques

- For a given GPU device, several standard recipes exist to increase one of the performance measures (hardware occupancy, etc.)
- One effective technique is to let the kernel code depend on a parameter like the number of coefficients computed by a thread in a simulation or stencil computation.
- This strategy makes even more sense when the code is not targeting a specific GPU device, but a range thereof.

Parametric CUDA kernels: definition

Input

- A GPU device with hardware parameters like the size Z of the memory of an SM, the maximum number R of registers per threads, etc
- A for-loop nest where some of the outermost for-loops can be executed as parallel for-loops and the loop index ranges depend on parameters $n, m, p, \ldots$ called *data parameters*.
- Program parameters $k, \ell, \ldots$

Parametric CUDA kernels: definition

Input

- A GPU device with hardware parameters like the size Z of the memory of an SM, the maximum number R of registers per threads, etc
- A for-loop nest where some of the outermost for-loops can be executed as parallel for-loops and the loop index ranges depend on parameters $n, m, p, \ldots$ called *data parameters*.
- Program parameters $k, \ell, \ldots$

Output

- Pairs $(C_1, K_1), \ldots, (C_e, K_e)$ where each of $C_1, \ldots, C_e$ is a system of constraints (equations and inequalities) on $Z, R, \ldots, n, m, p, \ldots$, $k, \ell, \ldots$ and each of $K_1, \ldots, K_e$ is an SIMT code such that
  1. for each $i = 1 \cdots e$, for each value of $(Z, R, \ldots, n, m, p, \ldots, k, \ell, \ldots)$ satisfying C_i, the kernel executes correctly on the specified GPU device;
  2. for each value of $n, m, p, \ldots$ that make the loop nest execute correctly, there exists $1 \leq i \leq e$ and values of $k, \ell, \ldots$ such that $(Z, R, \ldots, n, m,$ $p, \ldots,$ $k, \ell)$ satisfy C_i and K_i produces the same output as the loop-nest.

Parametric CUDA kernels: toy example

- Hardware parameters: maximum number R of registers per threads, maximum number T of threads per thread-block, all other hardware limits being ignored.
- Program parameter: B the number of threads per thread-block.
- For loop best with one parameter N:

```
for  (int i=0; i<N; i++)
          a[i] = b[i] + c[i];
```

- The parametric kernel code consists of $(C1, K1), (C2, K2)$ where:

$$C1 : \begin{cases} B \leq T \\ R \leq 8 \end{cases}$$

```
__global__  void K1(int *a, int *b, int *c,
                              int N, int B)
{
     int i = blockIdx.x * B + threadIdx.x;
     if (i < N)
        a[i] = b[i] + c[i];
}


K1 <<<N/B, B>>> (a,b,c,N,B)
```

$$C2 : \begin{cases} B \leq T \\ R > 8 \end{cases}$$

```
__global__  void K2(int *a, int *b, int *c,
                          int halfN, int B)
{
     int i = blockIdx.x * B + threadIdx.x;
     int j = i + halfN;
     if (i < halfN)
        a[i] = b[i] + c[i];
        a[j] = b[j] + c[j];
}
halfN = N/2;
K2 <<<halfN/B, B>>> (a,b,c,halfN/B,B)
```

Parametric CUDA kernels: remarks

Advantages

- Hardware parameters do not need to be known at kernel code generation time.
- Once the kernel code is installed on the targeted hardware, hardware parameters are known and the kernel code is specialized at those values.
- Then, program parameters can be determined by auto-tuning techniques so as to optimize code performance.
- Of course, the selected kernel code K_i and the corresponding constraint system C_i narrow the search.

Parametric CUDA kernels: remarks

Advantages

- Hardware parameters do not need to be known at kernel code generation time.
- Once the kernel code is installed on the targeted hardware, hardware parameters are known and the kernel code is specialized at those values.
- Then, program parameters can be determined by auto-tuning techniques so as to optimize code performance.
- Of course, the selected kernel code K_i and the corresponding constraint system C_i narrow the search.

Main challenges

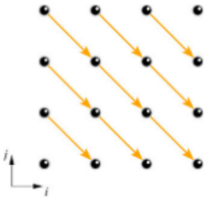
- Choosing which quantity can be a program parameter is a programer decision; typical choice: thread-block formats
- Generating the parametric kernel codes from the loop nest requires non-linear equation handling more precisely: quantifier elimination.

Plan

Automatic parallelization: plain multiplication

Serial dense univariate polynomial multiplication

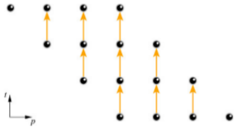
```
for(i=0; i<=n; i++){
  c[i] = 0; c[i+n] = 0;
  for(j=0; j<=n; j++)
    c[i+j] += a[i] * b[j];
}
```



Dependence analysis suggests to set $t(i,j) = n - j$ and $p(i,j) = i + j$.

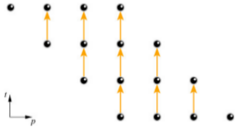
Asynchronous parallel dense univariate polynomial multiplication

```
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 ] ;
}
```



Generating parametric code & use of tiling techniques (1/2)

```
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 ] ;
}
```



Improving the parallelization

- The above generated code is not practical for multicore implementation: the number of processors is in $\Theta(n)$. (Not to mention poor locality!) and the work is unevenly distributed among the workers.
- We group the virtual processors (or threads) into 1D blocks, each of size B. Each thread is known by its block number b and a local coordinate u in its block.
- Blocks represent good units of work which have good locality property.
- This yields the following constraints: $0 \leq u < B,\ p = bB + u$.

Generating parametric code: using tiles (2/2)

We apply `RegularChains:-QuantifierElimination` on the left system (in order to get rid off i, j) leading to the relations on the right:

$$\left\{\begin{array}{c} o < n \\ 0 \le i \le n \\ 0 \le j \le n \\ t = n - j \\ p = i + j \\ 0 \le b \\ o \le u < B \\ p = bB + u, \end{array}\right. \qquad \left\{\begin{array}{c} B > 0 \\ n > 0 \\ 0 \le b \le 2n/B \\ 0 \le u < B \\ 0 \le u \le 2n - Bb \\ p = bB + u, \end{array}\right. \tag{1}$$

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++) {
    parallel_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 ];
   }
}
```

Plan

Preliminary implementation

- The *Polyhedral Parallel Code Generator* (PPCG) is a source-to-source framework performing C- to-CUDA automatic code generation. PPCG does not use parameters for the generated kernel code.
- Our MetaFork C-to-CUDA translator is based on PPCG In fact, we are currently modifying the PPCG framework to take parameters into account. Hence, our implementation is preliminary.

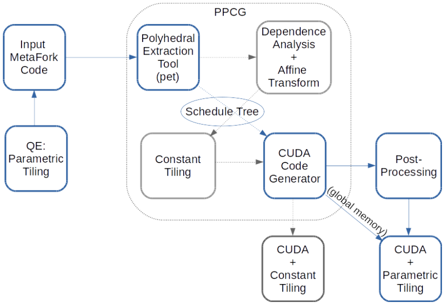


Figure: Components of METAFORK-to-CUDA generator of parametric code.

Reversing an array

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: Reversing a one-dimensional array

1D Jacobi

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: 1D-Jacobi

Matrix matrix 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: Matrix multiplication

LU decomposition

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: LU decomposition

Plan

Concluding remarks

Observations

- Most computer programs that we write are far to make an efficient use of the targeted hardware
- CUDA has brought supercomputing to the desktop computer, but is hard to optimize even to expert programmers.
- High-level models for accelerator programming, like OpenACC, OpenCL and MetaFork are an important research direction.

Our current work

- MetaFork-to-CUDA generates kernels depending on program parameters.
- This is feasible thanks to techniques from symbolic computation.
- Implementation is preliminary; yet experimental results are promising.
- We still need to better integrate MetaFork-to-CUDA into the PPCG framework (work in progress).

Our project web sites

Basic Polynomial Algebra Subprograms

www.bpaslib.org

www.metafork.org

www.cumodp.org

www.regularchains.org