Automatic Parallelization of Computer Programs: From C to CUDA

Marc Moreno Maza

University of Western Ontario, Canada

TRICS seminar
University of Western Ontario
March 2, 2016

Overview

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).

Overview

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

- Automatic generation of GPU code from C code is particularly challenging since CUDA programs combines two forms of parallelism (fork-join and SIMD).
- In addition, it is desirable that the generated GPU code depends on parameters (thread-block size, memory sizes, etc.) so as to improve portability and efficiency.

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.

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();
    }
}
```

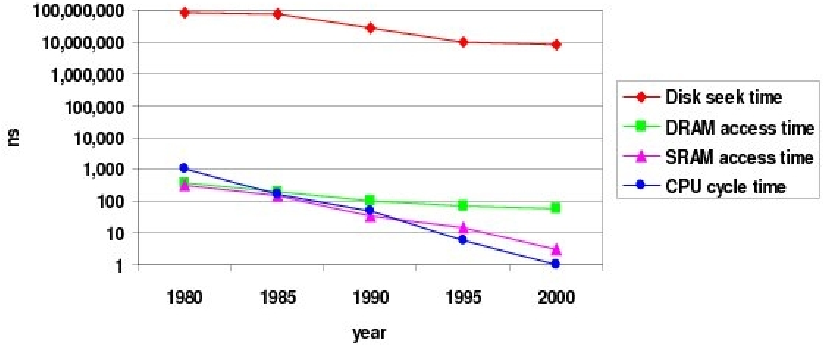
CUDA kernel corresponding to the first loop nest.

Plan

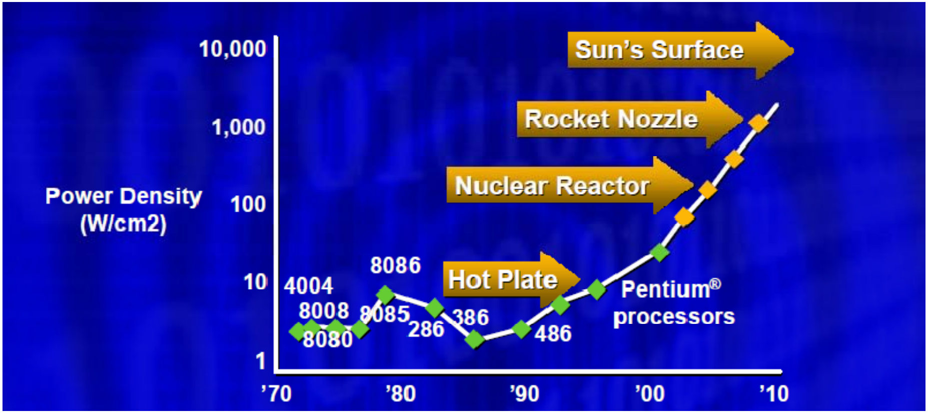
Plan

The CPU-Memory Gap

The increasing gap between DRAM, disk, and CPU speeds.

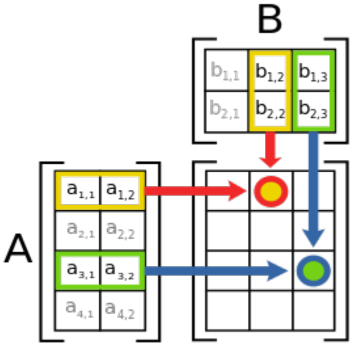


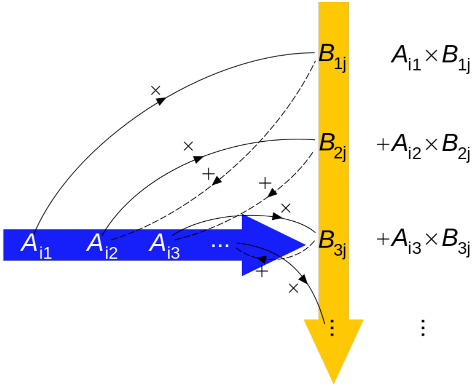
Once upon a time, everything was slow in a computer.



The second space race . . .

A driving application: matrix multiplication





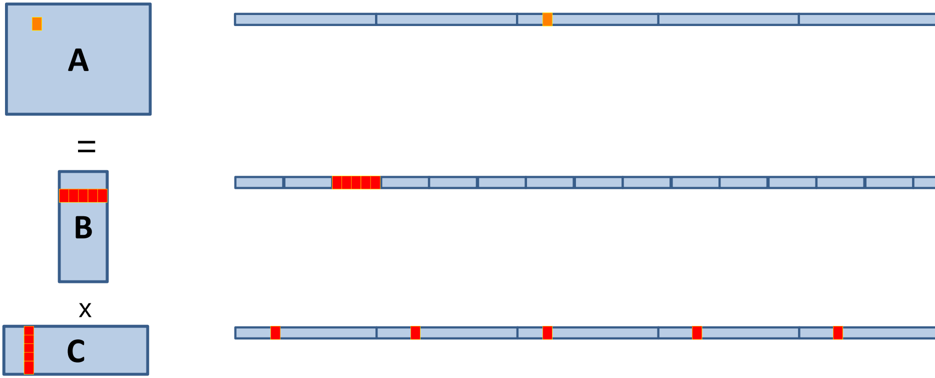
What’s wrong with my matrix multiplication code?

```
// x, y, z are positive integers

A = (double *)malloc(sizeof(double)*x*y);
B = (double *)malloc(sizeof(double)*x*z);
C = (double *)malloc(sizeof(double)*y*z);

// A, B, C encode dense matrices in row-major layout

..........................................

for (i = 0; i < x; i++)
  for (j = 0; j < y; j++)
     for (k = 0; k < z; k++)
            A[ i ][ j ] += B[ i ][ k ] * C[ k ][ j ];
```

High cache miss rate

```
A = (double *)malloc(sizeof(double)*x*y);
B = (double *)malloc(sizeof(double)*x*z);
C = (double *)malloc(sizeof(double)*y*z);

.........................................

for (i = 0; i < x; i++)
  for (j = 0; j < y; j++)
     for (k = 0; k < z; k++)
           A[ i ][ j ] += B[ i ][ k ] * C[ k ][ j ];
```

For z large enough, one cache miss per flop: poor spatial data locality!



Discontinuous accesses in `C` yields high cache miss rate.

A better program

```
 A = (double *)malloc(sizeof(double)*x*y);
 B = (double *)malloc(sizeof(double)*x*z);
 C = (double *)malloc(sizeof(double)*y*z);
 Cx = (double *)malloc(sizeof(double)*y*z);

 .........................................

for(j =0; j < y; j++)
   for(k=0; k < z; k++)
     Cx [ j ][ k ] = C[ k ][ j ] ;
 for (i = 0; i < x; i++)
   for (j = 0; j < y; j++)
      for (k = 0; k < z; k++)
        A[ i ][ j ] += B[ i ][ k ] * Cx[ j ][ k ];
```

What's wrong with my program again?

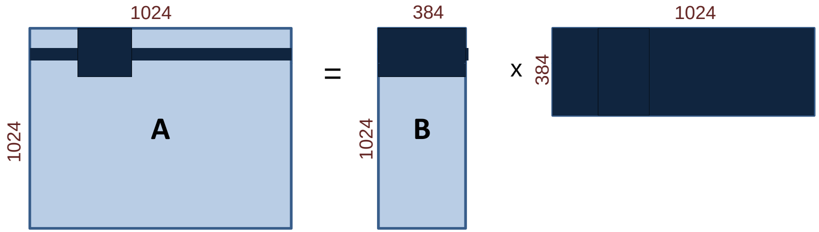
```
 A = (double *)malloc(sizeof(double)*x*y);
 B = (double *)malloc(sizeof(double)*x*z);
 C = (double *)malloc(sizeof(double)*y*z);
 Cx = (double *)malloc(sizeof(double)*y*z);

 .........................................

for(j =0; j < y; j++)
   for(k=0; k < z; k++)
     Cx [ j ][ k ] = C[ k ][ j ] ;
 for (i = 0; i < x; i++)
   for (j = 0; j < y; j++)
      for (k = 0; k < z; k++)
         A[ i ][ j ] += B[ i ][ k ] * Cx[ j ][ k ];
```

Still high cache miss rate!

```
 A = (double *)malloc(sizeof(double)*x*y);
 B = (double *)malloc(sizeof(double)*x*z);
 C = (double *)malloc(sizeof(double)*y*z);
 Cx = (double *)malloc(sizeof(double)*y*z);


 .........................................


for(j =0; j < y; j++)
   for(k=0; k < z; k++)
     Cx [ j ][ k ] = C[ k ][ j ] ;
for (i = 0; i < x; i++)
   for (j = 0; j < y; j++)
      for (k = 0; k < z; k++)
          A[ i ][ j ] += B[ i ][ k ] * Cx[ j ][ k ];
```

For computing each row (resp. column) of `A` we read the corresponding row (resp. column) of `B` (resp. `C`) `y` (resp. `x`) times: poor temporal data locality!

Issues with data reuse



- Naive calculation of a row of A, so computing 1024 coefficients: 1024 accesses in A, 384 in B and $1024 \times 384 = 393,216$ in C. Total $= 394,524$.
- Computing a 32×32-block of A, so computing again 1024 coefficients: 1024 accesses in A, 384×32 in B and 32×384 in C. Total $= 25,600$.
- The iteration space is traversed so as to reduce memory accesses.

A better program

```
 A = (double *)malloc(sizeof(double)*x*y);
 B = (double *)malloc(sizeof(double)*x*z);
 C = (double *)malloc(sizeof(double)*y*z);
 Cx = (double *)malloc(sizeof(double)*y*z);

 .........................................

for(j =0; j < y; j++)
   for(k=0; k < z; k++)
      Cx [ j ][ k ] = C[ k ][ j ] ;

 for(j =0; j < y; j++)
   for(k=0; k < z; k++)
     IND(Cx,j,k,z) = IND(C,k,j,y);
 for (i = 0; i < x; i += BLOCK_X)
   for (j = 0; j < y; j += BLOCK_Y)
     for (k = 0; k < z; k += BLOCK_Z)
        for (i0 = i; i0 < min(i + BLOCK_X, x); i0++)
          for (j0 = j; j0 < min(j + BLOCK_Y, y); j0++)
            for (k0 = k; k0 < min(k + BLOCK_Z, z); k0++)
               A[ i0 ][ j0 ] += B[ i0 ][ k0 ] * Cx[ j0 ][ k0 ];
```

For well chosen values of BLOCK_X, BLOCK_Y, BLOCK_Z, this program is cache-complexity optimal among all dense matrix multiplication algorithms with cubic running time.

What's wrong with my program again?

```
A = (double *)malloc(sizeof(double)*x*y);
B = (double *)malloc(sizeof(double)*x*z);
C = (double *)malloc(sizeof(double)*y*z);
Cx = (double *)malloc(sizeof(double)*y*z);

.........................................

for(j =0; j < y; j++)
   for(k=0; k < z; k++)
      Cx [ j ][ k ] = C[ j ][ k ] ;

for(j =0; j < y; j++)
  for(k=0; k < z; k++)
     IND(Cx,j,k,z) = IND(C,k,j,y);
for (i = 0; i < x; i += BLOCK_X)
  for (j = 0; j < y; j += BLOCK_Y)
     for (k = 0; k < z; k += BLOCK_Z)
       for (i0 = i; i0 < min(i + BLOCK_X, x); i0++)
         for (j0 = j; j0 < min(j + BLOCK_Y, y); j0++)
           for (k0 = k; k0 < min(k + BLOCK_Z, z); k0++)
               A[ i0 ][ j0 ] += B[ i0 ][ k0 ] * Cx[ j0 ][ k0 ];
```

My program is serial and all my computers have multicore processors

```
 A = (double *)malloc(sizeof(double)*x*y);
 B = (double *)malloc(sizeof(double)*x*z);
 C = (double *)malloc(sizeof(double)*y*z);
 Cx = (double *)malloc(sizeof(double)*y*z);

 .........................................

for(j =0; j < y; j++)
   for(k=0; k < z; k++)
      Cx [ j ][ k ] = C[ k ][ j ] ;

for(j =0; j < y; j++)
  for(k=0; k < z; k++)
     IND(Cx,j,k,z) = IND(C,k,j,y);
for (i = 0; i < x; i += BLOCK_X)
  for (j = 0; j < y; j += BLOCK_Y)
     for (k = 0; k < z; k += BLOCK_Z)
       for (i0 = i; i0 < min(i + BLOCK_X, x); i0++)
         for (j0 = j; j0 < min(j + BLOCK_Y, y); j0++)
           for (k0 = k; k0 < min(k + BLOCK_Z, z); k0++)
              A[ i0 ][ j0 ] += B[ i0 ][ k0 ] * Cx[ j0 ][ k0 ];
```

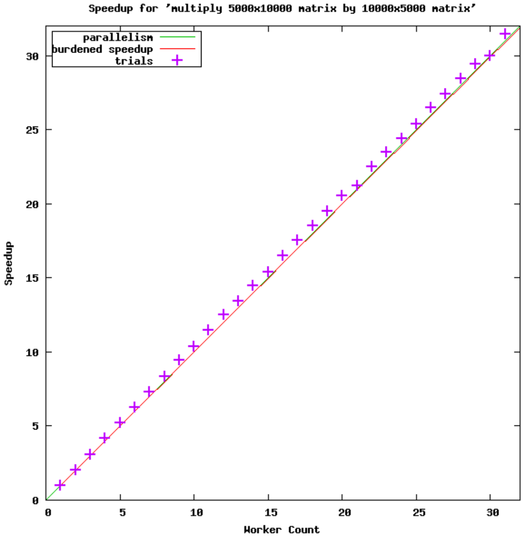
A nearly optimal program for parallel dense cubic matrix multiplication on multicore architectures

```
void parallel_dandc(int i0, int i1, int j0, int j1, int k0, int k1, int* A, int lda, int* B, int ldb, int* C, int ldc, int

{
  int di = i1 - i0;
  int dj = j1 - j0;
  int dk = k1 - k0;
  if (di >= dj && di >= dk && di >= X) {
    int mi = i0 + di / 2;
  cilk_spawn parallel_dandc(i0, mi, j0, j1, k0, k1, A, lda, B, ldb, C, ldc,X);
    parallel_dandc(mi, i1, j0, j1, k0, k1, A, lda, B, ldb, C, ldc,X);
  cilk_sync;
 } else if (dj >= dk && dj >= X) {

    int mj = j0 + dj / 2;
   cilk_spawn  parallel_dandc(i0, i1, j0, mj, k0, k1, A, lda, B, ldb, C, ldc,X);
    parallel_dandc(i0, i1, mj, j1, k0, k1, A, lda, B, ldb, C, ldc,X);
   cilk_sync;
  } else if (dk >= X) {

    int mk = k0 + dk / 2;
    parallel_dandc(i0, i1, j0, j1, k0, mk, A, lda, B, ldb, C, ldc,X);
    parallel_dandc(i0, i1, j0, j1, mk, k1, A, lda, B, ldb, C, ldc,X);

  } else {
    mm_loop_serial2(C, k0, k1,  A, i0, i1, B, j0, j1, lda)  ;
   /* for (int i = i0; i < i1; ++i)
      for (int j = j0; j < j1; ++j)
        for (int k = k0; k < k1; ++k)
          Ci * ldc + j += Ai * lda + k * Bk * ldb + j;*/
  }
}
```

Is that all?

```
void parallel_dandc(int i0, int i1, int j0, int j1, int k0, int k1, int* A, int lda, int* B, int ldb, int* C, int lc

{
    int di = i1 - i0;
    int dj = j1 - j0;
    int dk = k1 - k0;
    if (di >= dj && di >= dk && di >= X) {
        int mi = i0 + di / 2;
    cilk_spawn parallel_dandc(i0, mi, j0, j1, k0, k1, A, lda, B, ldb, C, ldc,X);
        parallel_dandc(mi, i1, j0, j1, k0, k1, A, lda, B, ldb, C, ldc,X);
    cilk_sync;
  } else if (dj >= dk && dj >= X) {

        int mj = j0 + dj / 2;
     cilk_spawn   parallel_dandc(i0, i1, j0, mj, k0, k1, A, lda, B, ldb, C, ldc,X);
        parallel_dandc(i0, i1, mj, j1, k0, k1, A, lda, B, ldb, C, ldc,X);
     cilk_sync;
    } else if (dk >= X) {

        int mk = k0 + dk / 2;
        parallel_dandc(i0, i1, j0, j1, k0, mk, A, lda, B, ldb, C, ldc,X);
        parallel_dandc(i0, i1, j0, j1, mk, k1, A, lda, B, ldb, C, ldc,X);

    } else {
        mm_loop_serial2(C, k0, k1,  A, i0, i1, B, j0, j1, lda)  ;
       /* for (int i = i0; i < i1; ++i)
            for (int j = j0; j < j1; ++j)
                for (int k = k0; k < k1; ++k)
                    Ci * ldc + j += Ai * lda + k * Bk * ldb + j;*/
    }
}
```

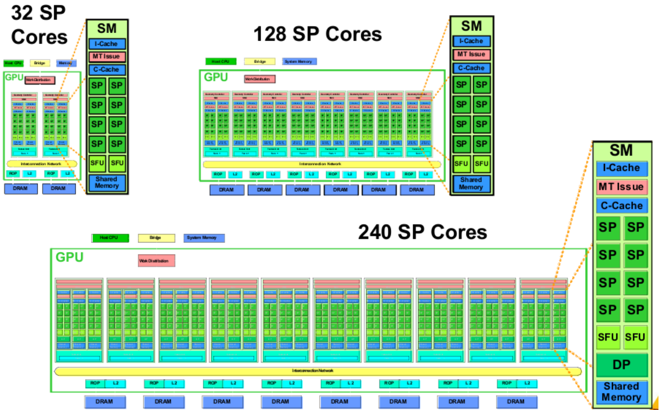
Benchmarks for the parallel version of the cache-efficient mm



Plan

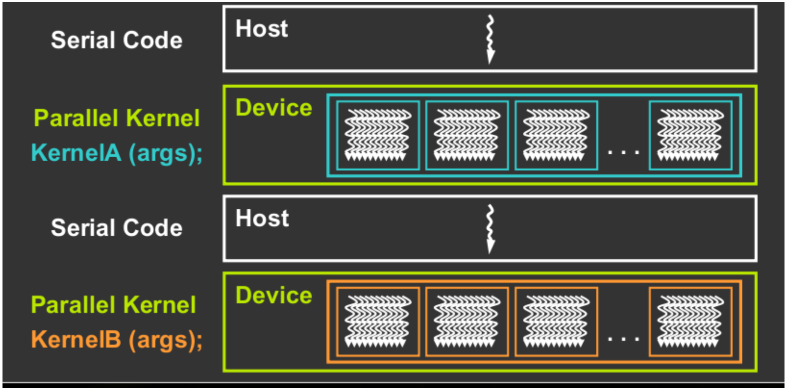
CUDA design goals

- Enable heterogeneous systems (i.e., CPU+GPU)
- Scale to 100's of cores, 1000's of parallel threads
- Use C/C++ with minimal extensions
- Let programmers focus on parallel algorithms (well, that's the intention)

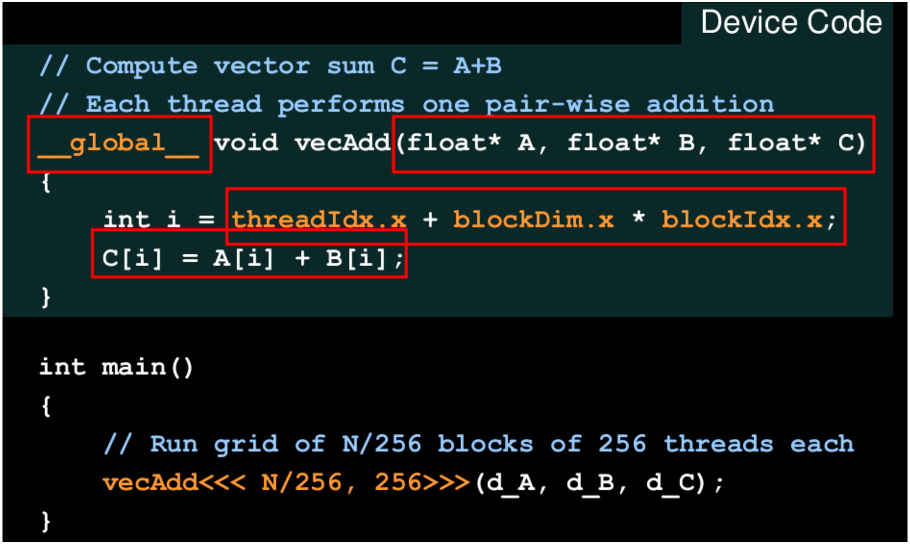


Heterogeneous programming

- A CUDA program is a serial program with parallel kernels, all in C.
- The serial C code executes in a host (= CPU) thread
- The parallel kernel C code executes in many device threads across multiple GPU processing elements, called streaming processors (SP).

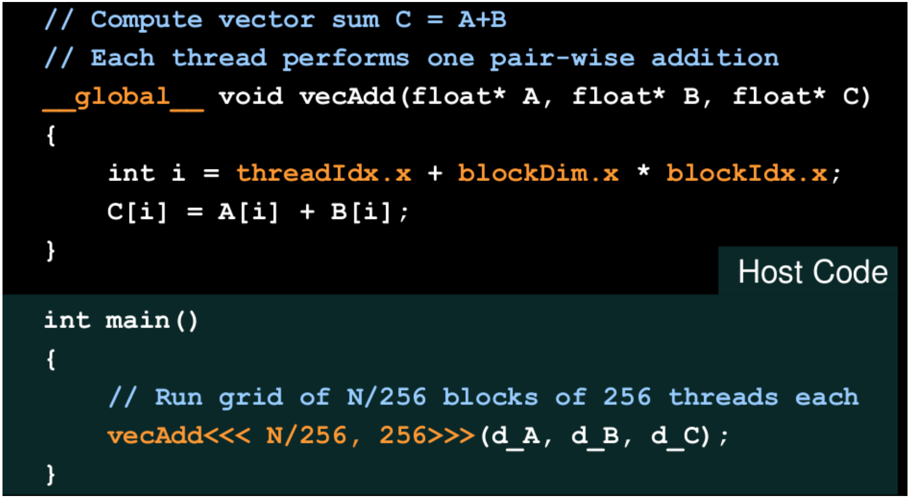


Vector addition on GPU (1/4)



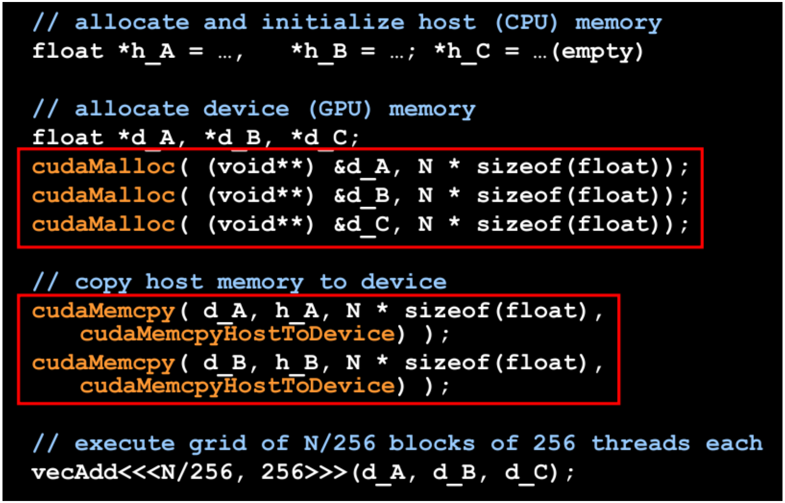
```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Vector addition on GPU (2/4)



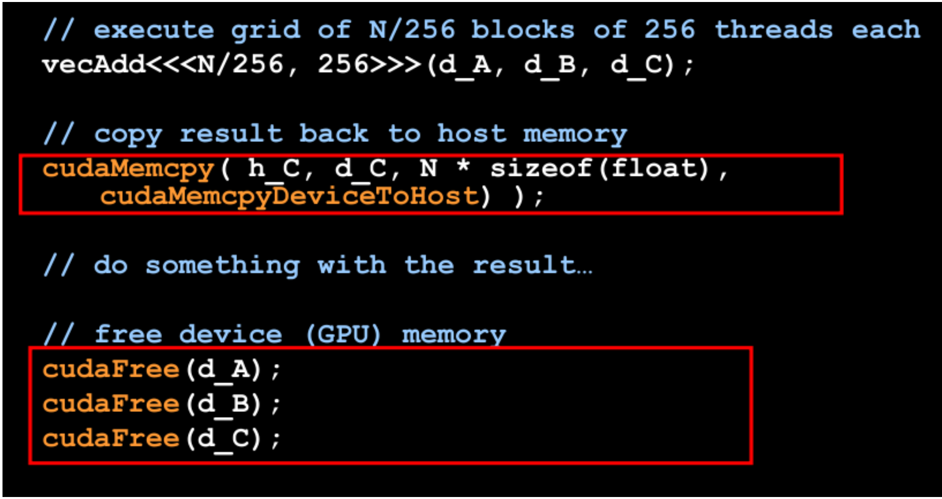
```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Vector addition on GPU (3/4)



```
// allocate and initialize host (CPU) memory
float *h_A = …,    *h_B = …; *h_C = …(empty)

// allocate device (GPU) memory
float *d_A, *d_B, *d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float),
   cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float),
   cudaMemcpyHostToDevice) );

// execute grid of N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

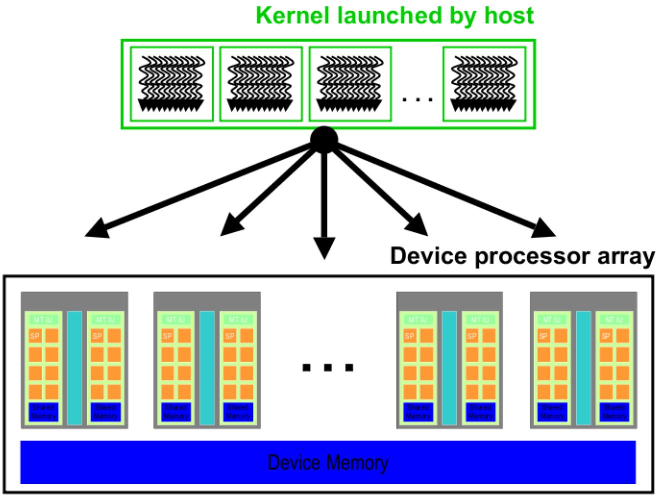
Vector addition on GPU (4/4)



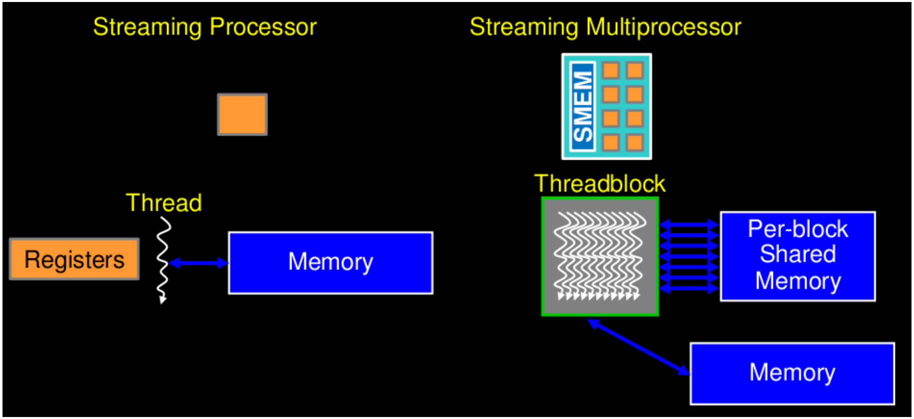
```
// execute grid of N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);

// copy result back to host memory
cudaMemcpy( h_C, d_C, N * sizeof(float),
    cudaMemcpyDeviceToHost) );

// do something with the result…

// free device (GPU) memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Blocks run on multiprocessors

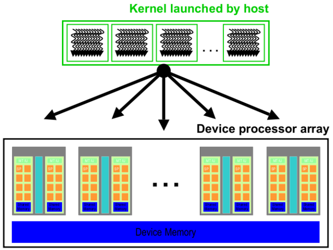
Hardware allocates resources to blocks and schedules threads.



Streaming processors and multiprocessors



Performance measures of CUDA kernels



- 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 compared to the hardware limit).

Plan

Loop transformation and automatic parallelization

Parallélisation à l'ancienne

```
for(i=0; i<=n-1; i++){                  par_for(i=0; i<=n-1; i++){
  c[i] = 0.0;                             c[i] = 0.0;
  for(j=0; j<=n-1; j++)        ⇒          for(j=0; j<=n-1; j++)
    c[i] += a[i][j]*b[j];                   c[i] += a[i][j]*b[j];
}                                       }
```

The parallel code is (restricted to be) isomorphic to the serial code.

Parallélisation polyhédrique

```
for (n=1; n<51; n++)                for (t=2; t<51; t++)
  for (k=1; k<51-n; k++) ⇒            par_for (p=1; p<t; p++)
    c[n,k] = c[n-1,k]                       c[t-p, p] = c[-1+t-p, p]
           + c[n,k-1];                                + c[t-p, p-1];
```

Generating the parallel code requires a "good" change of coordinates.
Here $t = n + k, p = k$, where t means *time* and p means *processor*.

Automatic parallelization: principles

Dependence Analysis: The sequential input program, is transformed to a geometrical object in *the index space.* For most programs, the loop indices satisfy a system of affine inequalities, thus defining a *polyhedron* in the index space. Then, one determines a partition of the iteration space such that within a part, iterations do not depend on each other (by considering memory accesses to shared variables).

Parallelization: The source index space (with loop variables as coordinates) is mapped to a target time-space (where time and processor are coordinates). Techniques of linear programming transform the original polyhedron into a new one in the target space.

Code Generation: From the target polyhedron, one deduces a loop nest where some loops are parallel. In order to obtain efficient code, scheduling, data locality and parallelism overheads lead to further transformations (exchanging loops, introducing new coordinates such as in blocked-matrix multiplication.)

Automatic parallelization: people

Bib refs

- Pioneer works: (Leslie Lamport 1974) (Allen and Kennedy, 1984)
- Introduction of the *Polyhedral Model* by Paul Feautrier in the early 90's. Largely extended by his students and their students (F. Irigoin, P. Boulet, C. Bastoul, etc.) and other research groups.
- Concurrently to Feautrier's group, other research teams have further developed the model: Christian Lengauer and his followers (M. Griebl, A. Grösslinger, etc.) P. (Saday) Sadayappan and Uday Bondhugula at Ohio State Univ., S. Rajopadhy at Colorado State Univ., J. (Ram) Ramanujam at Louisiana State Univ.
- This talk follows and responds to a 2006 JSC paper by A. Grösslinger, M. Griebl and C. Lengauer.

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.

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

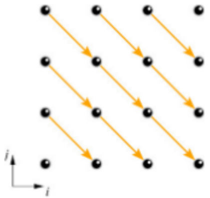
A driving application: matrix multipliication

$$
\begin{array}{rcrcrcrcrcr}
 & & & & & & 4x^3 & - & 3x & + & 6 \\
\times & & & & & & 2x^2 & - & 4x & - & 3 \\
\hline
 & & & - & 12x^3 & & & + & 9x & - & 18 \\
 & - & 16x^4 & & & + & 12x^2 & - & 24x & & \\
8x^5 & & & - & 6x^3 & + & 12x^2 & & & & \\
\hline
8x^5 & - & 16x^4 & - & 18x^3 & + & 24x^2 & - & 15x & - & 18
\end{array}
$$

Automatic parallelization: plain multiplication

Serial dense univariate polynomial multiplication

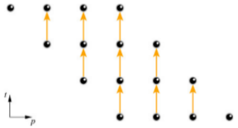
```
for(i=0; i<=n; i++){
  c[i] = 0; c[i+n] = 0;
  for(j=0; j<=n; j++)
    c[i+j] += a[i] * b[j];
}
```



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

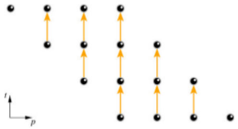
Asynchronous parallel dense univariate polynomial multiplication

```
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: $0 \leq u < B, \ p = bB + u$.

Quantifier elimination

Example

- Given a system of equation and inequalities (please note the non-linear one) involving variables n, i, j, p, t, u, b

$$\begin{cases} o < n \\ 0 \leq i \leq n \\ 0 \leq j \leq n \\ t = n - j \\ p = i + j \\ 0 \leq b \\ o \leq u < B \\ \mathbf{p = bB + u}, \end{cases} \tag{1}$$

  determine for which values n, p, t, u, b the above system has solutions (thus eliminating i, j).
- Techniques developed in our team and implemented in `RegularChains` library `www.regularchains.org` yield

$$\begin{cases} B > 0 \\ n > 0 \\ \mathbf{0 \leq b \leq 2n/B} \\ 0 \leq u < B \\ \mathbf{0 \leq u \leq 2n - Bb} \\ \mathbf{p = bB + u}, \end{cases} \tag{2}$$

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:

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

From where we derive the following program:

```
for (p=0; p<=2*n; p++) c [ p ] = 0;
parallel_for (b=0; b<= 2 n / B; b++) {
    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 ];
   }
}
```

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.

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.

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.

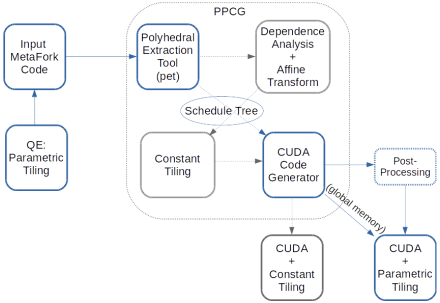


Figure: Components of METAFORK-to-CUDA generator of parametric code.

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 (1/2)

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.

Concluding remarks (2/2)

Our current work

- MetaFork-to-CUDA generates kernels depending on program parameters (like number of threads per block) and machine parameters (like shared memory size) are allowed.
- This is feasible thanks to techniques from quantifier elimination (QE).
- Machine parameters and program parameters can be respectively determined and optimized, once the generated code is installed on the target machine.
- The optimization part can be done from numerical computation and/or auto-tuning.
- Our implementation is very preliminary; yet experimental results are promising
- We still need to better integrate MetaFork-to-CUDA into the PPCG framework: make a better use of their internals and avoid duplicating work (when robustifying the code).

Team members and students involved

Current students

Parisa Alvandi, Haowei Chen, **Xiaohui Chen**, Egor Chesakov, Yiming Guan, Davood Mohajerani, Mahsa Kazemi, Robert Moir, Steven Thornton **Ning Xie**.

Alumni

Moshin Ali (ANU, Australia), Jinlong Cai (Microsoft), **Changbo Chen (CIGIT)**, **Sardar Anisula Haque (Geomechanica)** Zunaid Haque (IBM), François Lemaire (U. Lille 1, France), Xin Li (U. Carlos III, Spain), Wei Pan (Intel), Paul Vrbik (U. Newcastle, Australia), Yuzhen Xie (COT), Li Zhang (IBM) . . .

Our project web sites

Basic Polynomial Algebra Subprograms

www.bpaslib.org

MetaFork

www.metafork.org

www.cumodp.org

www.regularchains.org