Quantifier Elimination, Polyhedral Computations and their Applications to the Parallelization of Computer Programs

Marc Moreno Maza[1,2,3]
joint work with
Changbo Chen[1] Xiaohui Chen[2,3] Abdoul-Kader Keita[4]
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

East Coast Computer Algebra Day,
The Fields Institute, 3 October, 2015.

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!

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!

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;*/
    }
}
```

Plan

Plan

Automatic parallelization: objectives

Background

- Automatic generation of parallel code from serial one is a dramatically needed: most of our computers are parallel machines, but most of the programs that we have or write are serial ones.
- Automatic generation of parallel code is very hard and in many cases hopeless, but makes sense for kernels in scientific computing (dense linear and polynomial algebra, stencil computations).

Automatic parallelization: objectives

Background

- Automatic generation of parallel code from serial one is a dramatically needed: most of our computers are parallel machines, but most of the programs that we have or write are serial ones.
- Automatic generation of parallel code is very hard and in many cases hopeless, but makes sense for kernels in scientific computing (dense linear and polynomial algebra, stencil computations).

C to CUDA

- Automatic parallelization of simple C code to GPU code (with parameters such as block size, number of processors, etc.) can improve code development substantially.
- 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.

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 input loop nest is transformed to a geometrical object (generally 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.

Parallelization

The source index space (with loop variables as coordinates) is mapped to a target time-space (where time and processor are coordinates).

Code Generation

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

```
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.

A complete example: Jacobi

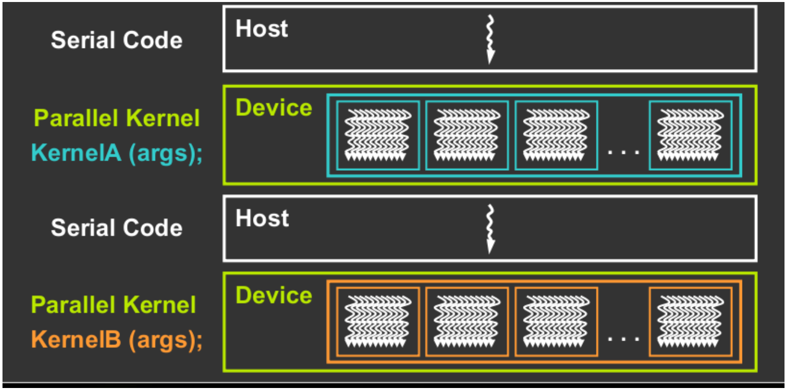
```
#include "jacobi_kernel.hu"
__global__ void kernel0(int *a, int *b, int N,
              int T, int ub_v, int B, int c0)
{
    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    int private_p;
    int private_y;
    int private_z;
    extern __shared__ int shared_a[];

    #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();
      for (int c2 = t0; c2 < B; c2 += 512) {
        private_p = ((((c1) * (B)) + (c2)) + 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.

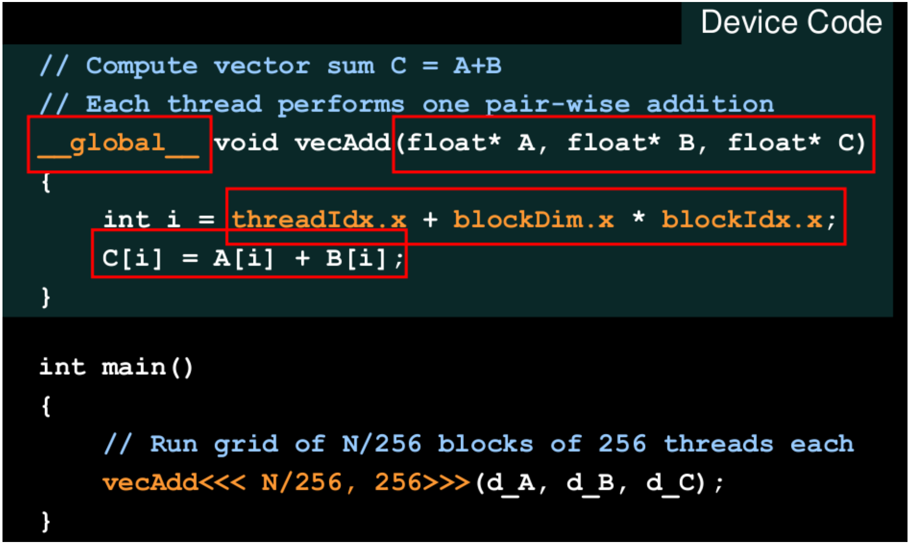
Plan

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).

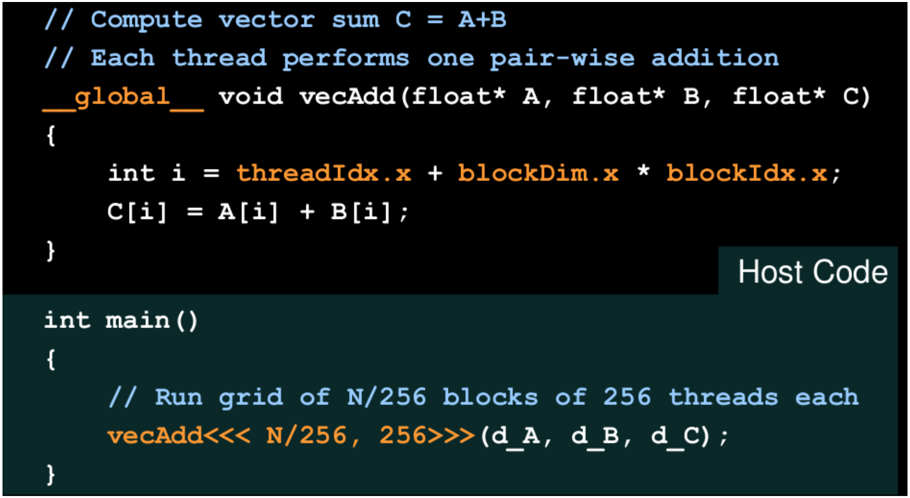


Vector addition on GPU (1/4)



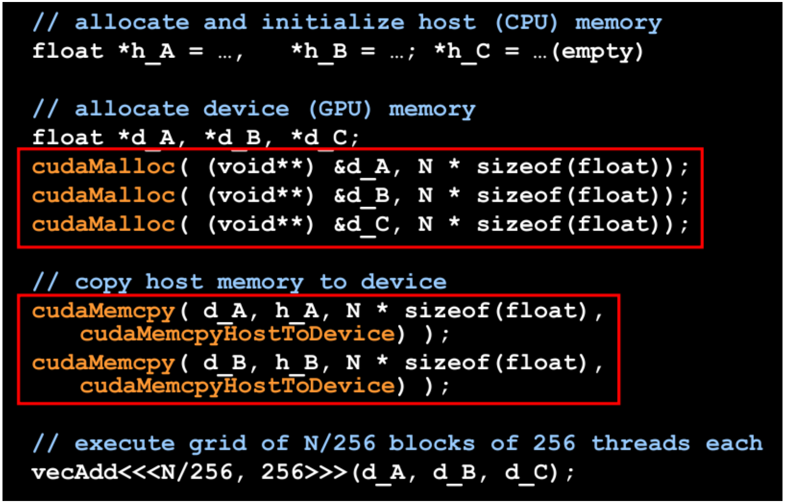
```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

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)



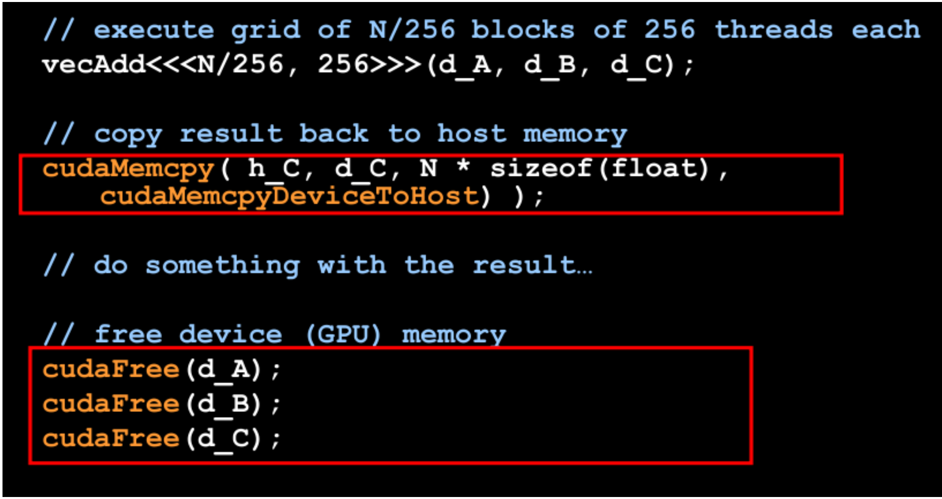
```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

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)



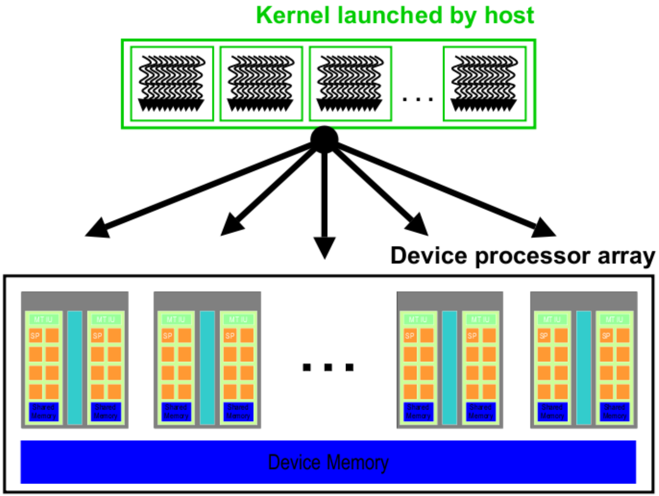
```
// allocate and initialize host (CPU) memory
float *h_A = …,    *h_B = …; *h_C = …(empty)

// 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)

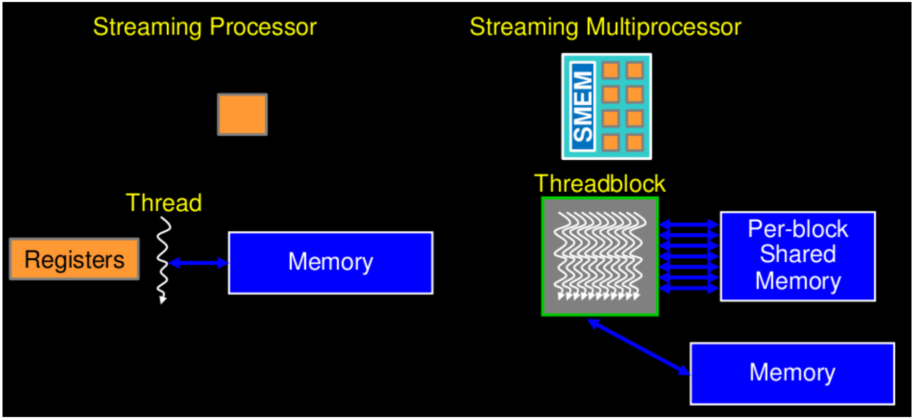


```
// execute grid of N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);

// 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

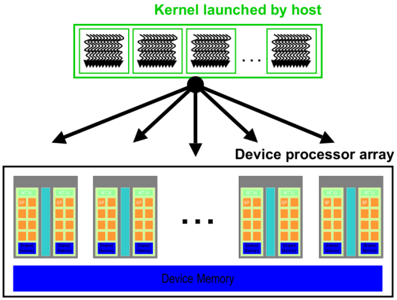


Streaming processors and multiprocessors



Blocks run on multiprocessors: four principles

Hardware allocates resources to blocks and schedules threads.



The user should:

1. expose as much parallelism as possible,
2. optimize memory usage for maximum bandwidth,
3. maximize occupancy to hide latency,
4. optimize instruction usage for maximum throughput.

Plan

The Many-Core Machine (MCM) model: the abstract machine

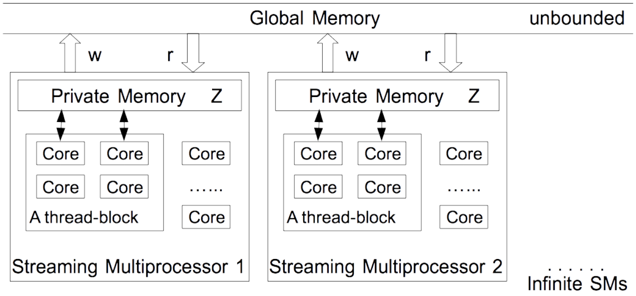


Figure: A many-core machine

- Global memory has high latency and low throughput while private memories have low latency and high throughput.

(Sardar A. Haque, M. & Ning XIe, ParCo 2015)

The Many-Core Machine (MCM) model: program structure

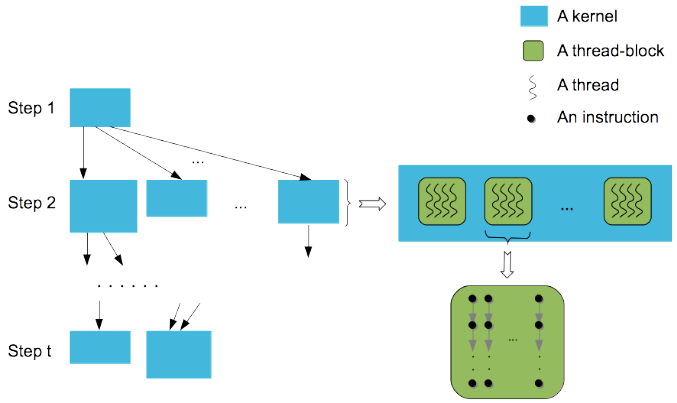


Figure: Sketch of a many-core machine program

The MCM model: parameters and complexity measures

Machine parameters

- Z: Private memory size of any streaming multiprocessor (SM),
- U: Data transfer time.

Program parameters

- ℓ: number of threads per thread-block,
- number of data words read/written per thread,
- ...

Complexity measures

- The **work** accounts for the total amount of local operations (ALU and private memory accesses)
- The **span** accounts for the maximum number of local operations along a path in the DAG representing the program
- The **parallelism overhead** for the total amount of data transfer.

Fast Fourier Transform

Let f be a vector with coefficients in a field (either a prime field like $\mathbb{Z}/p\mathbb{Z}$ or $\mathbb{C}$) and size n, which is a power of 2. Let ω be a n-th primitive root of unity.

The n-point *Discrete Fourier Transform* (DFT) at ω is the linear map defined by $x \longmapsto \mathrm{DFT}_n\, x$ with

$$\mathrm{DFT}_n = [\omega^{ij}]_{0 \leq i,\, j < n}.$$

We are interested in comparing popular algorithms for computing DFTs on many-core architectures:

- Cooley & Tukey FFT algorithm,
- Stockham FFT algorithm.

Fast Fourier Transforms: Cooley & Tukey vs Stockham

The work, span and parallelism overhead ratios between Cooley & Tukey and Stockham FFT algorithms are, respectively,

$$\frac{\mathrm{W}_{ct}}{\mathrm{W}_{sh}} \sim \frac{n\,(34\,\log_2(n)\,\log_2(\ell) + 47\,\log_2(n) + 333 - 136\log_2(\ell))}{43\,n\,\log_2(n) + \frac{n}{4\,\ell} + 12\,\ell + 1 - 30\,n},$$

$$\frac{\mathrm{S}_{ct}}{\mathrm{S}_{sh}} \sim \frac{34\,\log_2(n)\,\log_2(\ell) + 47\,\log_2(n) + 2223 - 136\log_2(\ell)}{43\,\log_2(n) + 16\,\log_2(\ell) + 3},$$

$$\frac{\mathrm{O}_{ct}}{\mathrm{O}_{sh}} = \frac{2\,n\,U\,(\frac{4\,\log_2(n)}{\ell} + \log_2(\ell) - \frac{\log_2(\ell)+15}{\ell})}{5\,n\,U\,(\frac{\log_2(n)}{\ell} + \frac{1}{4\,\ell})},$$

where ℓ is the number of threads per thread-block.

- Both the work and span of the algorithm of Cooley & Tukey are increased by $\Theta(\log_2(\ell))$ factor w.r.t their counterparts in Stockham algorithm.

Fast Fourier Transforms: Cooley & Tukey vs Stockham

The ratio $R = T_{ct}/T_{sh}$ of the estimated running times (using our Graham-Brent theorem) on $\Theta(\frac{n}{\ell})$ SMs is [1]:

$$R \sim \frac{\log_2(n)(2\,U\,\ell + 34\,\log_2(\ell) + 2\,U)}{5\,\log_2(n)\,(U + 2\,\log_2(\ell))},$$

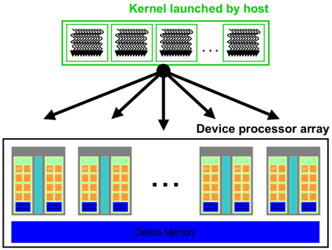
when n escapes to infinity. This latter ratio is greater than 1 if and only if $\ell > 1$.

n	Cooley & Tukey	Stockham
2^{14}	0.583296	0.666496
2^{15}	0.826784	0.7624
2^{16}	1.19542	0.929632
2^{17}	2.07514	1.24928
2^{18}	4.66762	1.86458
2^{19}	9.11498	3.04365
2^{20}	16.8699	5.38781

Table: Running time (secs) with input size n on GeForce GTX 670.

[1] ℓ is the number of threads per thread-block.

A popular performance counter: occupancy



- The **occupancy** of an SM is $A_{\mathrm{warp}}/M_{\mathrm{warp}}$, where A_{warp} and M_{warp} are respectively the number of active warps and maximum number of running warps per SM.
- Warps require resources (registers, shared memory, thread slots) to run
- A_{warp} is bounded over by $M_{\mathrm{block}}\,\ell$, where M_{block} is the maximum number of active blocks.
- Hence a small value for ℓ may limit occupancy.
- But larger ℓ will reduce the amount of registers and shared memory

Summary: we need parametric CUDA kernels

Overall, both theoretical models and empirical performance counters suggest that:

1. Generating kernel code depending on ℓ and other parameters (like the number of coefficients written and/or computed by a thread) is an important code optimization technique.
2. Once the machine parameters (like S_{warp}, M_{warp}, M_{block}, Z) are known, then optimizing at run-time the values of those program parameters (like ℓ and others) can be done by numerical methods and/or auto-tuning.

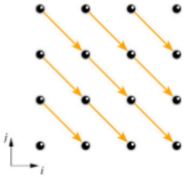
Questions?

Plan

Automatic parallelization: plain multiplication

Serial dense univariate polynomial multiplication

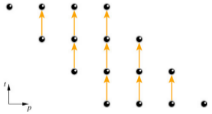
```
for(i=0; i<=n; i++){
  c[i] = 0; c[i+n] = 0;
  for(j=0; j<=n; j++)
    c[i+j] += a[i] * b[j];
}
```



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

Synchronous parallel dense univariate polynomial multiplication

```
for (p=0, p<=2*n, p++) c[p]=0;

for (t=0, t=n, t++)
  meta_for (p=n-t; p<=2*n -t; p++)
    c[p] = c[p] + a[t+p-n] * b[n-t];
}
```



Dependence analysis: how?

Problem

- Among all possible change of coordiantes, one typically wants to minimze cache complexity (or data communication).
- This translates to a problem of the form: Find the lexicographical minimum of $\{x \mid Ax + By \geq c, Dy \geq e\}$, where x, y, c, e are vectors and A, B, D are matrices all in integer coefficients.
- The result is a piecewise affine function.

Solutions

- Two popular solutions which both reduce to test whether a polyhedral set has integer points or not.
- One is based on a variant on the Simplex Algorithm (P. Feautrier) the other on a variant of Fourier-Motzkin Elimination (W. Pugh).
- In each case, the algorithm reduces to test whether a linear system $Ax = b$ has integer solutions,
- Which can be done by considering the *Hermite Normal Form* of A.
- The `PolyhedralSets` library has code for this task.

Interlude: Fourier-Motzkin Algorithm

Folklore

- A *polyhedral set* K of $\mathbb{R}^d$ is the solution set of a system of linear equations and inequalities. The projections of K to linear subspaces of $\mathbb{R}^d$ can be computed by the famous *Fourier-Motzkin Algorithm* (FMA), which is a natural adaptation of Gaussian Elimination.
- Most users believe that, if K is the intersection of n closed half-spaces of $\mathbb{R}^d$, then FMA runs in $O(n^{2^d})$ coefficient operations.

Interlude: Fourier-Motzkin Algorithm

Folklore

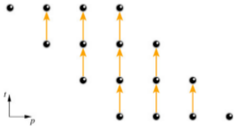
- A *polyhedral set* K of $\mathbb{R}^d$ is the solution set of a system of linear equations and inequalities. The projections of K to linear subspaces of $\mathbb{R}^d$ can be computed by the famous *Fourier-Motzkin Algorithm* (FMA), which is a natural adaptation of Gaussian Elimination.
- Most users believe that, if K is the intersection of n closed half-spaces of $\mathbb{R}^d$, then FMA runs in $O(n^{2^d})$ coefficient operations.

Reality

- In fact, this bound comes from (many) non-handled superfluous constraints generated by FMA.
- Those can be eliminated by LP (as suggested by L. Khachiyan, who gave no complexity estimates) and this works very well in practice.
- If the maximum absolute value of coefficients is bounded by T then FMA for projecting into a sub-coordinate space of co-dimension s can produce an irredundant system in $O\left(2^s\,(d+\delta^2)^{10}\log(d\,\delta\,T)\right)$ bit operations (with Rong Xiao).
- Application: Consider a system of equations and inequalities with a few non-linear monomials. We use tag variables to reduce FMA and recover the true result by CAD when unwrapping the tagged monomials.

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

```
meta_for (p=0; p<=2*n; p++){
  c[p]=0;
  for (t=max(0,n-p); t<= min(n,2*n-p
    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.

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:

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

From where we derive the following program:

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

Generation of parametric parallel programs

Summary

- Given a parallel algorithm (e.g. divide-and-conquer matrix multiplication) expressed in MetaFork with **program parameters** like B,
- given a type of hardware accelerators, like GPGPUs, characterized by **machine parameters**, like Z, M,
- we can:
  1. automatically generate code that depends on the machine and program parameters Z, M, $\ldots$, B, by means of **symbolic computation**,
  2. specialize the machine parameters for a specific accelerator of the above type,
  3. optimize the program parameters by means of **numerical computation**.

Note

The symbolic computation part, which is a special form of **quantifier elimination (QE)** (Changbo Chen & M^3, ISSAC 2014 & CASC 2015) is performed by our `RegularChains` library in Maple available at

`www.regularchains.org`

Plan

Another complete example: (simplified) LU

```
///////////////////////////////////////////////////////////////
// Main loop for serial LU
// Assumes: column major representation of L and U
//          thus, column major initialization of L and U from
//          the input matrix M.
// Assumes: n > 1.
///////////////////////////////////////////////////////////////
for (int k = 0; k < n; ++k) {
     // Update the k-th column of L
    for (int i = 0; i < n-k-1; i++) {
        int p = i + k + 1;
        L[k][p] = U[k][p] / U[k][k];
    |

    // Update the the last n-k columns of U
    for (int i = 0; i < n-k-1; i++) {
        int p = i + k + 1;
        for (int j = k; j < n; j++)
            U[j][p] -= L[k][p] * U[j][k];
    }
}
```

Original C code.

Another complete example: (simplified) LU

```
for (int k = 0; k < n; ++k) {
    // Update the k-th column of L
    int I_k = n-k-1;
    int J_k = n-k;
    meta_schedule {
        meta_for (int b=0; b < I_k / B; b++)
            meta_for (int u=0; u < B; u++) {
                int p = b * B + u + k + 1;
                if (p < n)
                    L[k][p] = U[k][p] / U[k][k];
            }

        // Update the last n-k columns of U
        meta_for (int b_j=0; b_j < J_k / Sqrt_T; b_j++)
            meta_for (int b_i=0; b_i < J_k / Sqrt_T; b_i++)
                meta_for (int u_j=0; u_j < Sqrt_T; u_j++)
                    meta_for (int u_i=0; u_i < Sqrt_T; u_i++) {
                        int i = b_i * Sqrt_T + u_i;
                        if (i < n-k-1) {
                           int j = b_j * Sqrt_T + u_j;
                           int q = j + k;
                           int p = i + k + 1;
                           U[q][p] -= L[k][p] * U[q][k];
            }
        }
    }
}
```

MetaFork code obtained via quantifier elimination.

Another complete example: (simplified) LU

```
#include "jacobi_kernel.hu"

__global__ void kernel1(double *L, double *U, int n, int dim, int B, int ub, int T, int T2, int c0)
{
    int b0 = blockIdx.y, b1 = blockIdx.x;
    int t0 = threadIdx.y, t1 = threadIdx.x;
    //int private_u;
    //int private_q;
    //extern __shared__ double shared_L[];
    extern __shared__ double shared_U[];

__shared__ double s_l[1];
__shared__ double s_u[BLOCK][BLOCK];

    #define floord(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
    #define min(x,y)    ((x) < (y) ? (x) : (y))
    for (int c1 = b1; c1 < dim; c1 += 256)
      for (int c2 = b0; c2 < B; c2 += 256)
        for (int c3 = 0; c3 < ub; c3 += 1) {
          if (!t0 && !t1)
            s_l[0] = L[c0 * (n) + c1 * (B) + c2];
          for (int c4 = t1; c4 < T; c4 += 32)
            for (int c5 = t0; c5 < T; c5 += 32)
              shared_U[c4 * (T) + c5] = U[((((c3) * (T2)) + ((c4) * (T))) + (c5)) * (n) + c1 * (B) + c2];
          for (int c4 = t1; c4 < T; c4 += 32)
            for (int c5 = t0; c5 < T; c5 += 32)
              s_u[c4][c5] = U[((((c3) * (T2)) + ((c4) * (T))) + (c5)) * (n) + c0];
          __syncthreads();
          for (int c4 = t1; c4 < T; c4 += 32)
            for (int c5 = t0; c5 < T; c5 += 32)
              if ((((((c0) + 1) - ((c1) * (B))) < (B)) && ((((-(B)) * (c1)) + (c0)) < (c2))) && ((c2) < ((n) - ((c1) * (B))))) {
                shared_U[c4 * (T) + c5] -= s_l[0] * s_u[c4][c5];
              }
          __syncthreads();
          for (int c4 = t1; c4 < T; c4 += 32)
            for (int c5 = t0; c5 < T; c5 += 32)
              U[((((c3) * (T2)) + ((c4) * (T))) + (c5)) * (n) + c1 * (B) + c2] = shared_U[c4 * (T) + c5];
          __syncthreads();
        }
}
```

CUDA kernel corresponding to the second 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.

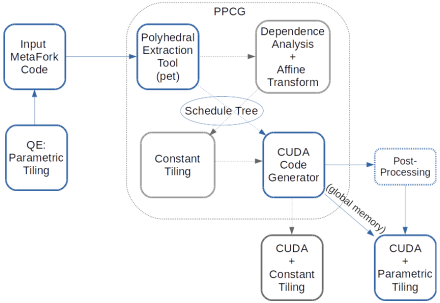


Figure: Components of MetaFork-to-CUDA generator of parametric code.

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).

Our project web sites

Basic Polynomial Algebra Subprograms

www.bpaslib.org

MetaFork

www.metafork.org

www.cumodp.org

www.regularchains.org