Comprehensive Optimization of Parametric Kernels for Graphics Processing Units

Xiaohui Chen[1], Marc Moreno Maza[2], Jeeva Paudel[3], Ning Xie[4]

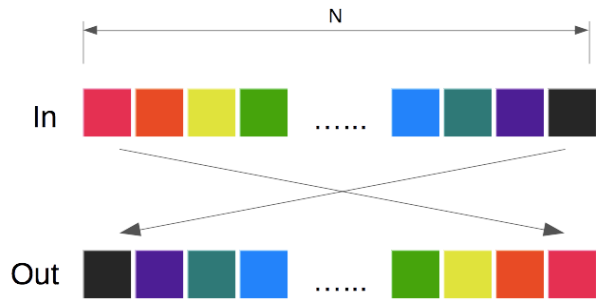
[1] AMD, Markham, Canada
[2] U. Western Ontario, London, Canada
[3] IBM Canada Ltd, Markham, Canada
[4] Huawei Technologies Canada, Markham, Canada

Applications of Computer Algebra
Session on High-Performance Computer Algebra
Jerusalem College of Technology, July 21, 2017

Automatic CUDA code generation (1/2)

Consider the following C program for reversing a one-dimensional array, whereas each thread can move one element of the input vector `In` to the corresponding position of the output vector `Out`.

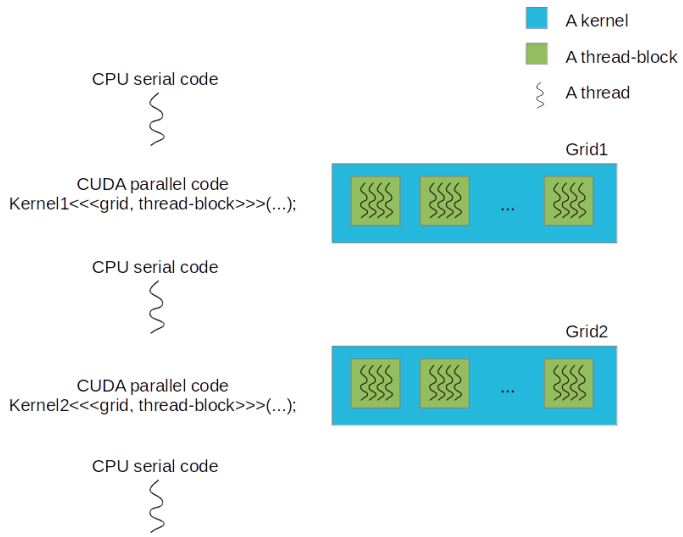
```
int N, In[N], Out[N];

// Initializing
for (int i = 0; i < N; i++)
  In[i] = i+1;

// Reversing the array In
for(int i = 0; i < N; i++)
  Out[N-1-i] = In[i];
```



Automatic CUDA code generation (2/2)

```
int N, In[N], Out[N];

// Initializing
for (int i = 0; i < N; i++)
  In[i] = i+1;

int *dev_In, *dev_Out;

// Allocating memory spaces on the device
cudaMalloc((void **) &dev_In, (N)*sizeof(int));
cudaMalloc((void **) &dev_Out, (N)*sizeof(int));

// Copying the data from host to device
cudaMemcpy(dev_In, In, (N)*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_Out, Out, (N)*sizeof(int), cudaMemcpyHostToDevice);

// Launching the kernel
dim3 dimBlock(32);
dim3 dimGrid(N/32);
kernel0 <<<dimGrid, dimBlock>>> (dev_In, dev_Out, N);

// Copying the data from device to host
cudaMemcpy(Out, dev_Out, (N)*sizeof(int), cudaMemcpyDeviceToHost);

// Freeing the memory spaces on the device
cudaFree(dev_In);
cudaFree(dev_Out);
```

The host code



The CPU-GPU co-processing programming

```
__global__ void kernel0(int *In, int *Out, int N) {
  int idx = blockIdx.x * 32 + threadIdx.x;
  __shared__ int shared_In[32];

  if (idx < N) {
    shared_In[threadIdx.x] = In[idx];
    __syncthreads();
    Out[N-1-idx] = shared_In[threadIdx.x];
  }
}
```

The device code

Related work

In popular projects (C-to-CUDA, PPCG, hiCUDA, CUDA-CHiLL),

- program parameters, like the number of threads per thread-block, and
- machine parameters, like the shared memory size,

are numerical values instead of unknown symbols.

- Muthu Manikandan Baskaran, J. Ramanujam and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Proceedings of CC'10/ETAPS'10*, pages 244-263, Berlin, Heidelberg, 2010. Springer-Verlag.
- Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Transactions on TACO*, 9(4):54, 2013.
- Tianyi D. Han and Tarek S. Abdelrahman. hiCUDA: A high-level directive based language for GPU programming. In *Proceedings of Workshop on GPGPU-2*, pages 52-61, 2009.
- Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen and Jacqueline Chame. A programming language interface to describe transformations and code generation. In *Proceedings of LCPC'10*, pages 136-150, Berlin, Heidelberg, 2011. Springer-Verlag.

Overview of our project

Objectives

We allow the generated CUDA code to depend on machine and program parameters. At code generation time:

- those parameters are treated as unknown symbols, and
- the generated code is optimized in the form of a case discussion, based on the possible values of those parameters.

Challenges

Non-linear polynomial expressions arise due to the symbolic parameters:

- techniques from symbolic computation support the necessary algebraic manipulation, but
- existing software tools for automatic code generation do not handle well non-linear polynomial expressions, say in dependence analysis or in computing schedules.

Comprehensive optimization: the array reversal example

```
__global__ void kernel1(int *In, int *Out,
                        int N, int B) {
  int idx = blockIdx.x * B + threadIdx.x;
  // BLOCK_0 should be pre-defined as
  // a constant and be equal to B
  __shared__ int shared_In[BLOCK_0];
  if (idx < N) {
    shared_In[threadIdx.x] = In[idx];
    __syncthreads();
    Out[N-1-idx] = shared_In[threadIdx.x];
  }
}

dim3 dimBlock(B);
dim3 dimGrid(N/B);
kernel1 <<<dimGrid, dimBlock>>>
              (dev_In, dev_Out, N, B);
```

Comprehensive optimization: the array reversal example

```
__global__ void kernel1(int *In, int *Out,
                        int N, int B) {
  int idx = blockIdx.x * B + threadIdx.x;
  // BLOCK_0 should be pre-defined as
  // a constant and be equal to B
  __shared__ int shared_In[BLOCK_0];
  if (idx < N) {
    shared_In[threadIdx.x] = In[idx];
    __syncthreads();
    Out[N-1-idx] = shared_In[threadIdx.x];
  }
}

dim3 dimBlock(B);
dim3 dimGrid(N/B);
kernel1 <<<dimGrid, dimBlock>>>
              (dev_In, dev_Out, N, B);
```

```
__global__ void kernel2(int *In, int *Out,
                                  int N, int B) {
  int even_idx = blockIdx.x*2*B+2*threadIdx.x;
  int odd_idx = blockIdx.x*2*B+2*threadIdx.x+1;
  // BLOCK_0 should be pre-defined as
  // a constant and be equal to B
  __shared__ int shared_In[2*BLOCK_0];
  if (even_idx < N && odd_idx < N) {
    shared_In[2*threadIdx.x] = In[even_idx];
    shared_In[2*threadIdx.x+1] = In[odd_idx];
    __syncthreads();
    Out[N-1-even_idx] = shared_In[2*threadIdx.x];
    Out[N-1-odd_idx] =
            shared_In[2*threadIdx.x+1];
  }
}

dim3 dimBlock(B);
dim3 dimGrid(N/(2*B));
kernel2 <<<dimGrid, dimBlock>>>
                  (dev_In, dev_Out, N, B);
```

Comprehensive optimization: the array reversal example

```
__global__ void kernel1(int *In, int *Out,
                        int N, int B) {
  int idx = blockIdx.x * B + threadIdx.x;
  // BLOCK_0 should be pre-defined as
  // a constant and be equal to B
  __shared__ int shared_In[BLOCK_0];
  if (idx < N) {
    shared_In[threadIdx.x] = In[idx];
    __syncthreads();
    Out[N-1-idx] = shared_In[threadIdx.x];
  }
}

dim3 dimBlock(B);
dim3 dimGrid(N/B);
kernel1 <<<dimGrid, dimBlock>>>
              (dev_In, dev_Out, N, B);
```

```
__global__ void kernel2(int *In, int *Out,
                                  int N, int B) {
  int even_idx = blockIdx.x*2*B+2*threadIdx.x;
  int odd_idx = blockIdx.x*2*B+2*threadIdx.x+1;
  // BLOCK_0 should be pre-defined as
  // a constant and be equal to B
  __shared__ int shared_In[2*BLOCK_0];
  if (even_idx < N && odd_idx < N) {
    shared_In[2*threadIdx.x] = In[even_idx];
    shared_In[2*threadIdx.x+1] = In[odd_idx];
    __syncthreads();
    Out[N-1-even_idx] = shared_In[2*threadIdx.x];
    Out[N-1-odd_idx] =
            shared_In[2*threadIdx.x+1];
  }
}

dim3 dimBlock(B);
dim3 dimGrid(N/(2*B));
kernel2 <<<dimGrid, dimBlock>>>
                  (dev_In, dev_Out, N, B);
```

$$C_1 : \begin{cases} \text{or} & B \le Z \\ & 6 \le R < 8 \end{cases} \qquad C_2 : \begin{cases} \text{and} & 2B \le Z \\ & 8 \le R \end{cases}$$

Z: maximum number of shared memory words per SM.

R: maximum number of registers per thread.

Plan

1. Generation of parametric `CUDA` kernels
   - The `MetaFork`-to-`CUDA` code generator
   - Experimentation
2. OpenMP 4.5 to MetaFork
3. Comprehensive Optimization of Parametric Kernels
4. Conclusion and future work

The MetaFork language and framework

```
long fib (long n) {
  if (n < 2) return n;
  else if (n < BASE)
    return fib_serial(n);
  else {
    long x, y;
    x = meta_fork fib(n-1);
    y = fib(n-2);
    meta_join;
    return x+y;
  }
}
```

```
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;
        b[p+1] = (a[p] + a[p+1] + a[p+2])/3;
      }
    meta_for (int v = 0; v < ub_v; v++)
      meta_for (int u = 0; u < B; u++) {
        int w = v * B + u;
        a[w+1] = b[w+1];
      }
  }
```

- extends C with 4 models of concurrency: fork-join, pipelining, device, à la MPI.
- provides translators between concurrency platforms: between CilkPlus and OpenMP, from MetaFork to CUDA
- fork-join keywords: meta_fork, meta_join and meta_for.
- SIMD-device: meta_schedule, cache
- The body of a meta_schedule statement is a sequence of meta_for loop nests and for loop nests.
- Each meta_for loop nest is executed by the device while for-loops are executed by host.
- Grid and thread-block formats are deduced from the ranges of the meta_for loop nest.

The MetaFork language

- The MetaFork framework[1] performs program translations between CilkPlus and OpenMP (both ways) targeting multi-cores.
- Extending C/C++: `meta_fork`, `meta_join` and `meta_for`.

```
long fib (long n) {
  if (n < 2) return n;
  else if (n < BASE)
    return fib_serial(n);
  else {
    long x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x+y;
  }
}
```

Original CilkPlus

```
long fib (long n) {
  if (n < 2) return n;
  else if (n < BASE)
    return fib_serial(n);
  else {
    long x, y;
    x = meta_fork fib(n-1);
    y = fib(n-2);
    meta_join;
    return x+y;
  }
}
```

Intermediate MetaFork

```
long fib (long n) {
  if (n < 2) return n;
  else if (n < BASE)
    return fib_serial(n);
  else {
    long x, y;
    #pragma omp task shared(x)
    x = fib(n-1);
    y = fib(n-2);
    #pragma omp taskwait
    return x+y;
  }
}
```

Translated OpenMP

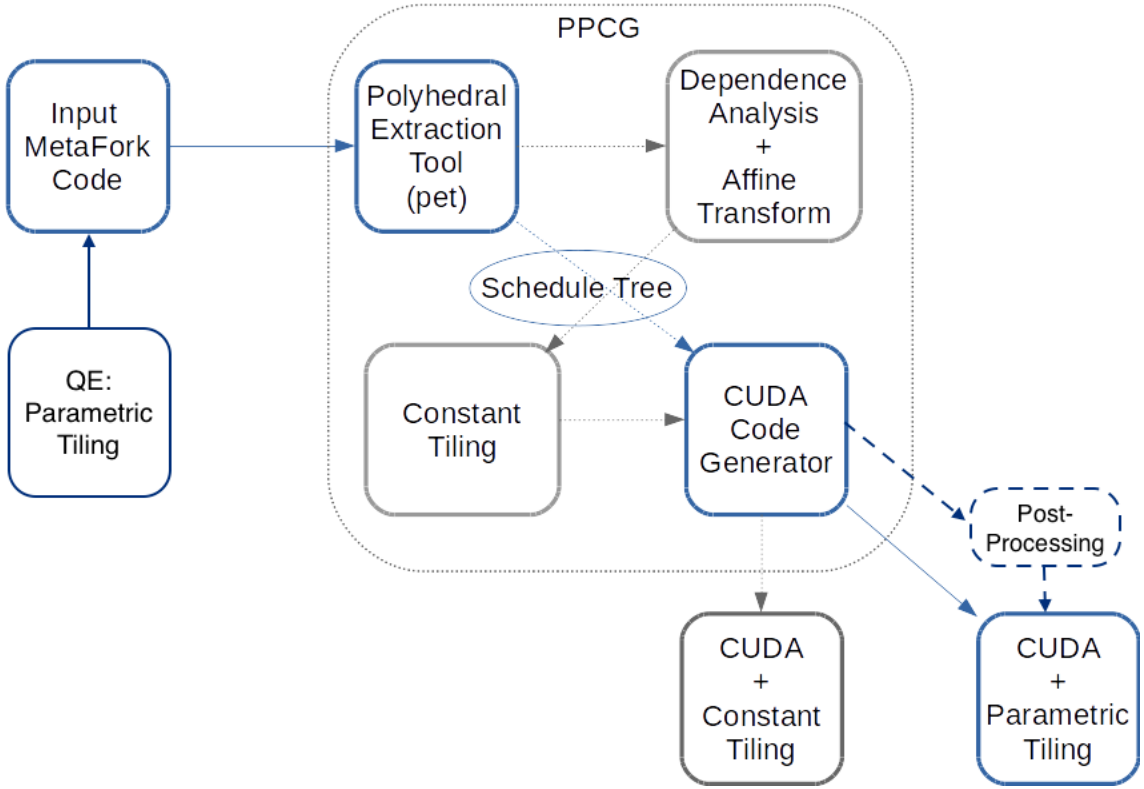
[1] Xiaohui Chen, Marc Moreno Maza, Sushek Shekar, and Priya Unnikrishnan. MetaFork: A framework for concurrency platforms targeting multicores. In *Proceedings of IWOMP'14*, pages 30-44, 2014.

meta_schedule statements

- The body of a meta_schedule statement is a sequence of meta_for loop nests and for loop nests. This body is converted to CUDA code
- Each meta_for loop nest is turned into a kernel call;
- Grid and thread-block formats are deduced from the loop counter ranges of the meta_for loop nest
- Tiling transformations can be done on each meta_for loop nest and support non-linear expressions in index arithmetic.

```
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;
        b[p+1] = (a[p] + a[p+1] + a[p+2])/3;
      }
    meta_for (int v = 0; v < ub_v; v++)
      meta_for (int u = 0; u < B; u++) {
        int w = v * B + u;
        a[w+1] = b[w+1];
      }
  }
}
```

The MetaFork-to-CUDA code generator



- The MetaFork-to-CUDA code generator[2] allows the automatic generation of kernels depending on parameters.
- This MetaFork-to-CUDA code generator modifies and extends PPCG[3], a C-to-CUDA compilation framework.

[2] Publicly available at http://www.metafork.org/

[3] PPCG's original code is available at https://www.openhub.net/p/ppcg.

Reversing a one-dimensional array

Serial code

```
for (int i = 0; i < N; i++)
  Out[N - 1 - i] = In[i];
```

MetaFork code

```
int ub_v = N / B;
meta_schedule {
  meta_for (int v = 0; v < ub_v; v++)
    meta_for (int u = 0; u < B; u++) {
      int inoffset = v * B + u;
      int outoffset = N - 1 - inoffset;
      Out[outoffset] = In[inoffset];
    }
}
```

Table: Speedup comparison between PPCG and MetaFork kernel code

Speedup (kernel)	Input size		
Thread-block size	2^{23}	2^{24}	2^{25}
PPCG			
32	8.312	8.121	8.204
MetaFork			
16	4.035	3.794	3.568
32	7.612	7.326	7.473
64	13.183	13.110	13.058
128	19.357	19.694	20.195
256	20.451	21.614	22.965
512	18.768	18.291	19.512

- Both MetaFork and PPCG generate CUDA code that uses a one-dimensional kernel grid and the shared memory.

LU decomposition

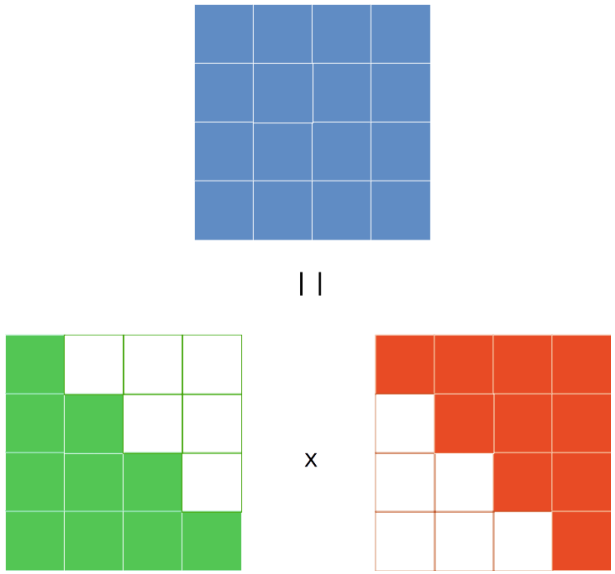
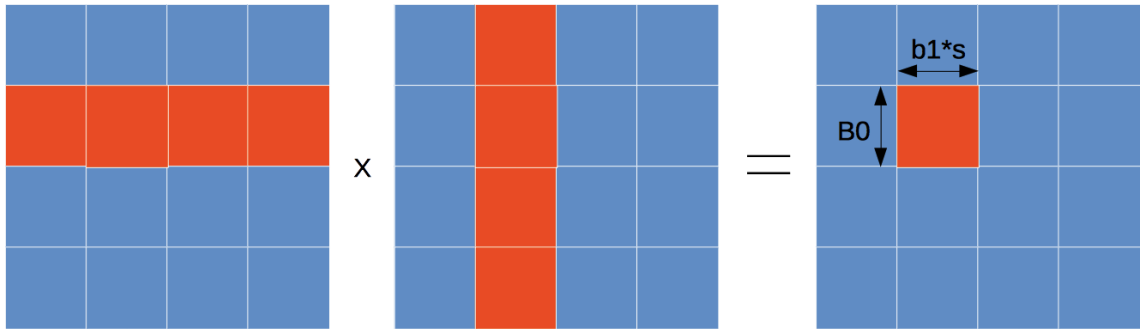


Table: Speedup comparison between PPCG and MetaFork kernel code

Speedup (kernel)				Input size	
Thread-block size kernel0, kernel1				$2^{10} * 2^{10}$	$2^{11} * 2^{11}$
PPCG					
32,	16	*	32	10.712	30.329
MetaFork					
128,	4	*	4	3.063	15.512
256,	4	*	4	3.077	15.532
512,	4	*	4	3.095	15.572
32,	8	*	8	10.721	37.727
64,	8	*	8	10.604	37.861
128,	8	*	8	10.463	37.936
256,	8	*	8	10.831	37.398
512,	8	*	8	10.416	37.840
32,	16	*	16	14.533	54.121
64,	16	*	16	14.457	54.034
128,	16	*	16	14.877	54.447
256,	16	*	16	14.803	53.662
512,	16	*	16	14.479	53.077

- MetaFork and PPCG both generate two CUDA kernels: one with a 1D grid and one with a 2D grid, both using the shared memory.

Matrix matrix multiplication (1/2)




```
// n * n matrices
// Program parameters: B0, b1, s
assert(BLOCK == min(B0, b1 * s));
int dim0 = n / B0, dim1 = n / (b1 * 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 / BLOCK; ++k)
        meta_for (int v = 0; v < B0; v++)
          meta_for (int u = 0; u < b1; u++)
            // Each thread computes BLOCK*s outputs
            for (int w = 0; w < s; ++w) {
              int p = i * B0 + v;
              int q = j * b1 * s + w * b1 + u;
              for (int z = 0; z < BLOCK; z++)
                c[p][q] += a[p][BLOCK*k+z] * b[BLOCK*k+z][q];
            }
}
```

Matrix matrix multiplication (2/2)

Table: Speedup factors obtained with kernels generated by `PPCG` and `MetaFork` with granularity, respectively, w.r.t. the serial C code with good data locality

Speedup (kernel)	Input size			
Thread-block size	$2^{10} * 2^{10}$		$2^{11} * 2^{11}$	
`PPCG`				
(16, 32)	109		105	
`MetaFork` with granularity				
	Granularity			
	2	4	2	4
(16, 4)	95	128	90	119
(32, 4)	128	157	125	144
(64, 4)	111	145	105	132
(8, 8)	131	151	126	146
(16, 8)	164	194	159	188
(32, 8)	163	187	158	202
(64, 8)	94	143	104	135

Plan

1. Generation of parametric CUDA kernels
2. OpenMP 4.5 to MetaFork
3. Comprehensive Optimization of Parametric Kernels
4. Conclusion and future work

LU decomposition

```
// Program parameters:
// G: grid size
// B: thread-block size

#define N 2048

for (int k = 0; k < N; ++k) {
 #pragma omp target
 #pragma omp teams num_teams(G) \
         thread_limit(B)
 #pragma omp distribute parallel \
         for schedule(static, 1)
  for (int i = 0; i < N; i++)
   if (i < N-k-1) {
    int p = i + k + 1;
    int in1 = k*N+p, in2 = k*N+k;
    L[in1] = U[in1] / U[in2];
    for (int j = k; j < N; j++) {
     int in3 = j*N+p;
     int in4 = k*N+p, in5 = j*N d+k;
     U[in3] -= L[in4] * U[in5];
    }
   }
}
```

```
for (int k = 0; k < N; ++k) {
 int tmp0 = (2048- 1 -0)/1 + 1;
 int tmp2 = G;
 int tmp3 = (tmp0- 1 +tmp2)/tmp2;
 int tmp5 = (tmp0- 1 -0)/tmp3 + 1;
 int tmp10 = (tmp3- 1 -0)/1 + 1;
 int tmp14 = B;
 meta_schedule {
  meta_for (int tmp6 = 0; tmp6 < tmp5; tmp6 += 1)
   meta_for (int tmp15 = 0; tmp15 < tmp14; tmp15 += 1)
    for (int tmp11 = tmp15; tmp11 < tmp10; tmp11 += tmp14)
     for (int tmp13 = 0; tmp13 < 1; tmp13 += 1) {
      int tmp4 = tmp6 * tmp3 + 0;
      int tmp9 = tmp11 * 1 + 0;
      int tmp8 = tmp13 * 1 + tmp9;
      int tmp1 = tmp8 * 1 + tmp4;
      int i = tmp1 * 1 + 0;
      if (i < N-k-1) {
       int p = i + k + 1;
       int in1 = k*N+p, in2 = k*N+k;
       L[in1] = U[in1] / U[in2];
       for (int j = k; j < N; j++) {
        int in3 = j*N+p;
        int in4 = k*N+p, in5 = j*N+k;
        U[in3] -= L[in4] * U[in5];
       }
      }
     }
 } }
```

Matrix matrix multiplication

```
// Program parameters:
// G: grid size
// B: thread-block size

#define N 2048

#pragma omp target \
        map(tofrom: C[0:N*N]) \
        map(to: A[0:N*N]) \
        map(to: B[0:N*N])
#pragma omp teams num_teams(G) \
        thread_limit(Bl)
#pragma omp distribute
for (i = 0; i < N; i++) {
 #pragma omp parallel \
         for schedule(static, 1)
 for (j = 0; j < N; j++) {
  float r_C;
  int idx1 = i*N+j;
  r_C = C[idx1];
  for (int k = 0; k < N; k++) {
   int idx2 = i*N+k, idx3 = k*N+j;
   r_C += A[idx2] * B[idx3];
  }
  C[idx1] = r_C;
 }
}
```

```
int tmp0 = (2048- 1 -0)/1 + 1;
int tmp4 = G;
int tmp5 = (tmp0- 1 +tmp4)/tmp4;
int tmp7 = (tmp0- 1 -0)/tmp5 + 1;
int tmp2 = (2048- 1 -0)/1 + 1;
int tmp12 = (tmp2- 1 -0)/1 + 1;
int tmp16 = Bl;
meta_schedule {
 meta_for (int tmp8 = 0; tmp8 < tmp7; tmp8 += 1)
  for (int tmp10 = 0; tmp10 < tmp5; tmp10 += 1)
   meta_for (int tmp17 = 0; tmp17 < tmp16; tmp17 += 1)
    for (int tmp13 = tmp17; tmp13 < tmp12; tmp13 += tmp16)
     for (int tmp15 = 0; tmp15 < 1; tmp15 += 1) {
      int tmp6 = tmp8 * tmp5 + 0;
      int tmp1 = tmp10 * 1 + tmp6;
      int i = tmp1 * 1 + 0;
      int tmp11 = tmp13 * 1 + 0;
      int tmp3 = tmp15 * 1 + tmp11;
      int j = tmp3 * 1 + 0;
      float r_C;
      int idx1 = i*N+j;
      r_C = C[idx1];
      for (int k = 0; k < N; k++) {
       int idx2 = i*N+k, idx3 = k*N+j;
       r_C += A[idx2] * B[idx3];
      }
      C[idx1] = r_C;
     }
}
```

Experimentation (1/3)

Vector addition with input size 4096*4096

Grid	Thread-block	Time (ms)
OpenMP 4.5-to-CUDA-via-MetaFork		
64	128	2.0099
64	256	1.6868
64	512	1.7670
64	1024	1.6030
128	128	1.7215
128	256	1.7654
128	512	1.6397
128	1024	1.5468
256	128	1.7769
256	256	1.6359
256	512	1.5733
256	1024	1.5359
512	128	1.6526
512	256	1.5914
512	512	1.5635
512	1024	1.5248
XLC/OpenMP 4.5 to CUDA		
512	1024	1.5863
128	1024	1.5869

Saxpy with input size 4096*64

Grid	Thread-block	Time (ms)
OpenMP 4.5-to-CUDA-via-MetaFork		
64	128	63.616
64	256	69.280
64	512	123.68
64	1024	196.42
128	128	27.296
128	256	37.664
128	512	53.632
128	1024	93.951
256	128	24.224
256	256	24.160
256	512	33.312
256	1024	58.816
512	128	22.592
512	256	23.392
512	512	27.904
512	1024	52.736
XLC/OpenMP 4.5 to CUDA		
512	128	32.448
128	1024	34.880

Experimentation (2/3)

1D Jacobi with input size 32768

Grid	Thread-block	Time (us)
OpenMP 4.5-to-CUDA-via-MetaFork		
64	128	12.9670
64	256	10.4190
64	512	12.0130
64	1024	15.9710
128	128	10.5060
128	256	11.5750
128	512	14.9460
128	1024	23.823
256	128	11.4970
256	256	14.5000
256	512	22.2330
256	1024	41.559
512	128	14.4670
512	256	21.5840
512	512	38.257
512	1024	76.943
XLC/OpenMP 4.5 to CUDA		
64	256	14.9140
128	1024	49.014

Matrix transpose with input size 16384

Grid	Thread-block	Time (ms)
OpenMP 4.5-to-CUDA-via-MetaFork		
64	128	391.81
64	256	392.00
64	512	400.56
64	1024	407.17
128	128	392.00
128	256	399.86
128	512	397.45
128	1024	396.19
256	128	398.81
256	256	395.27
256	512	399.73
256	1024	393.44
512	128	394.68
512	256	391.43
512	512	393.20
512	1024	393.49
XLC/OpenMP 4.5 to CUDA		
512	256	414.08
128	1024	463.62

Experimentation (3/3)

LU decomposition with input size 2048

Grid	Thread-block	Time
OpenMP 4.5-to-CUDA-via-MetaFork		
64	128	881.97us
64	256	879.51us
64	512	1.1664ms
64	1024	1.8814ms
128	128	745.48us
128	256	977.26us
128	512	1.5628ms
128	1024	2.6860ms
256	128	1.0022ms
256	256	1.5707ms
256	512	2.6665ms
256	1024	4.9606ms
512	128	1.6310ms
512	256	2.7041ms
512	512	4.9535ms
512	1024	9.4395ms
XLC/OpenMP 4.5 to CUDA		
128	128	752.47us
128	1024	2.6421ms

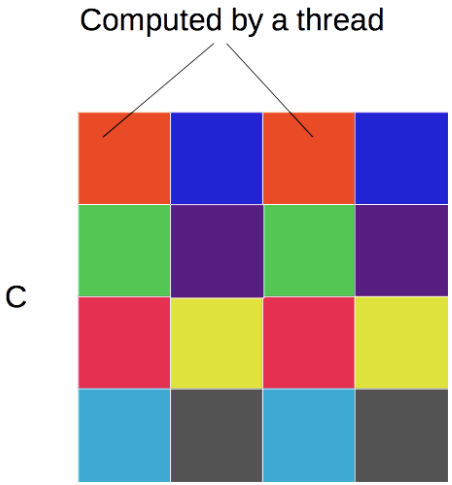
Matrix multiplication with input size 2048*2048

Grid	Thread-block	Time (ms)
OpenMP 4.5-to-CUDA-via-MetaFork		
64	128	251.32
64	256	233.32
64	512	302.42
64	1024	247.39
128	128	236.40
128	256	288.02
128	512	261.89
128	1024	213.77
256	128	321.43
256	256	265.28
256	512	224.83
256	1024	214.47
512	128	255.73
512	256	231.48
512	512	223.77
512	1024	217.44
XLC/OpenMP 4.5 to CUDA		
128	1024	342.37

Plan

1. Generation of parametric CUDA kernels
2. OpenMP 4.5 to MetaFork
3. Comprehensive Optimization of Parametric Kernels
   - Comprehensive optimization: input and output
   - Experimentation
4. Conclusion and future work

An example: Matrix addition (1/2)




```
int dim0 = N/B0, dim1 = N/(2*B1);
meta_schedule {
  meta_for (int v = 0; v < dim0; v++)
    meta_for (int p = 0; p < dim1; p++)
      meta_for (int u = 0; u < B0; u++)
        meta_for (int q = 0; q < B1; q++) {
          int i = v * B0 + u;
          int j = p * B1 + q;
          if (i < N && j < N/2) {
            c[i][j] = a[i][j] + b[i][j];
            c[i][j+N/2] =
               a[i][j+N/2] + b[i][j+N/2];
          }
        }
}
```

- Consider two machine parameters: the maximum number R_1 of registers per thread and the maximum number R_2 of threads per thread-block.
- The optimization strategy (w.r.t. register usage per thread) consists in reducing the work per thread via removing the 2-way loop unrolling.

An example: Matrix addition (2/2)

$$C_1 : \begin{cases} B_0 \times B_1 \leq R_2 \\ 14 \leq R_1 \end{cases}$$

```
__global__  void K1(int *a, int *b, int *c, int N,
                                    int B0, int B1) {
     int i = blockIdx.y * B0 + threadIdx.y;
     int j = blockIdx.x * B1 + threadIdx.x;
     if (i < N && j < N/2) {
        a[i*N+j] = b[i*N+j] + c[i*N+j];
        a[i*N+j+N/2] = b[i*N+j+N/2] + c[i*N+j+N/2];
     }
}
dim3 dimBlock(B1, B0);
dim3 dimGrid(N/(2*B1), N/B0);
K1 <<<dimGrid, dimBlock>>> (a, b, c, N, B0, B1);
```

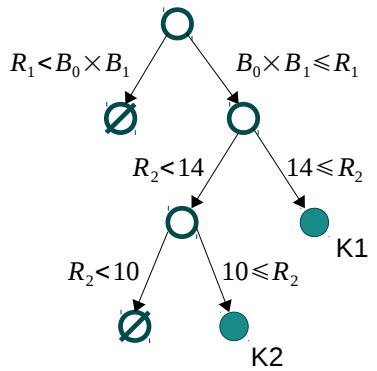
$$C_2 : \begin{cases} B_0 \times B_1 \leq R_2 \\ 10 \leq R_1 < 14 \end{cases}$$

```
__global__  void K2(int *a, int *b, int *c, int N,
                                    int B0, int B1) {
     int i = blockIdx.y * B0 + threadIdx.y;
     int j = blockIdx.x * B1 + threadIdx.x;
     if (i < N && j < N)
        a[i*N+j] = b[i*N+j] + c[i*N+j];
}
dim3 dimBlock(B1, B0);
dim3 dimGrid(N/B1, N/B0);
K2 <<<dimGrid, dimBlock>>> (a, b, c, N, B0, B1);
```

Algorithm

Matrix addition written in C

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



(S1) Tiling techniques, based on quantifier elimination (QE), are applied to the `for` loop nest in order to decompose the matrices into tiles of format $B_0 \times B_1$.

(S2) The tiled `MetaFork` code is mapped to an intermediate representation (IR) say that of LLVM, or alternatively, to PTX code.

(S3) Using this IR (or PTX) code, one can *estimate* the number of registers that a thread requires; thus, using LLVM IR on this example, we obtain an estimate of 14.

(S4) Next, we apply the optimization strategy, yielding a new IR (or PTX) code, for which register pressure reduces to 10. Since no other optimization techniques are considered, the procedure stops.

Comprehensive optimization

- Consider a code fragment $\mathcal{S}$ written in the C language.
- Let $R_1, \ldots, R_s$ the *hardware resource limits* of the targeted hardware device.
- Let $D_1, \ldots, D_u$ and $E_1, \ldots, E_v$ be the data parameters and program parameters of $\mathcal{S}$ (i.e scalar variable read but not written in $\mathcal{S}$)
- Let $P_1, \ldots, P_t$ be *performance measures* of a program running on the device.
- Functions to evaluate hardware and performance counters of $\mathcal{S}$
- Let $O_1, \ldots, O_w$ be strategies for optimizing resource consumption and performance; those are applied either to $\mathcal{S}$ or an IR (say PTX) of $\mathcal{S}$.

Algebraic systems $C_1, \ldots, C_e$ and corresponding programs $K_1, \ldots, K_e$ such that

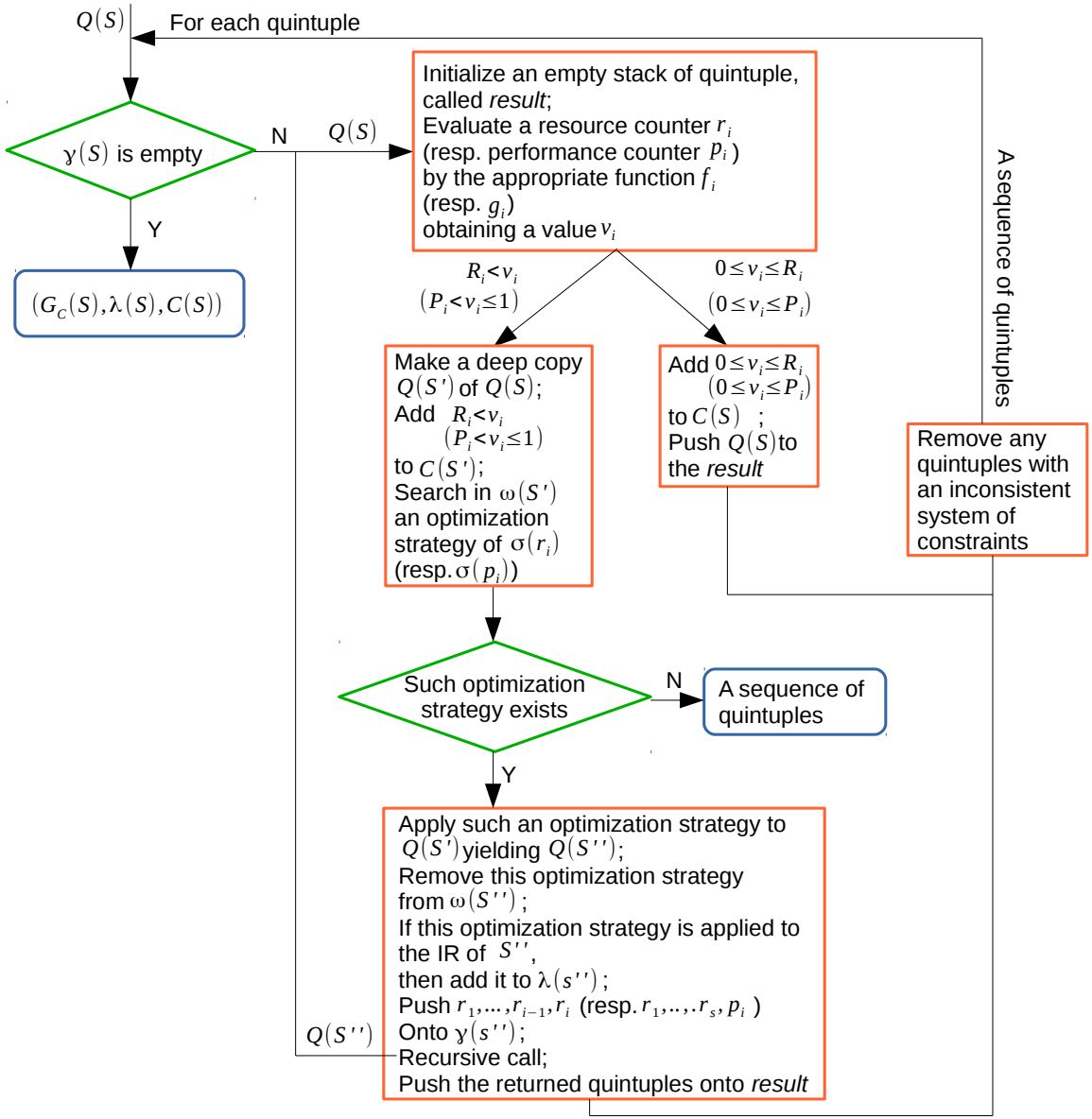
(i) [constraint soundness] each $C_1, \ldots, C_e$ is consistent

(ii) [code soundness] each K_i is a faithful translation of $\mathcal{S}$ whenever C_i holds

(iii) [coverage] for every run of $\mathcal{S}$, there exists a K_i producing the same result (under the conditions given by C_i)

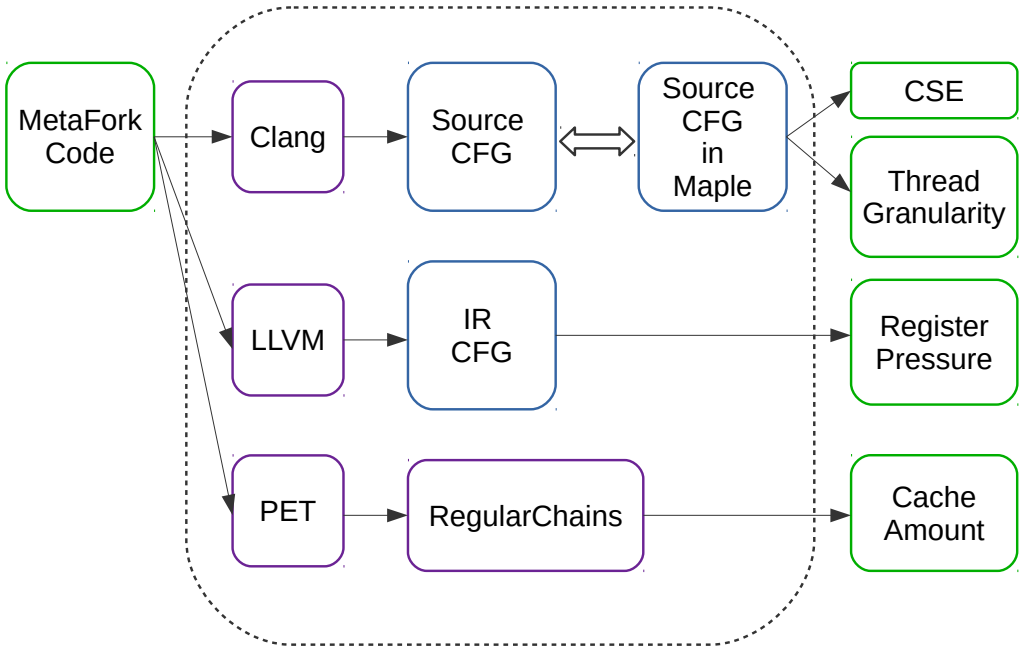
(iv) [optimality] every K_i (under the conditions given by C_i) optimizes at least one resource (performance) counter.

The algorithm



Implementation

- Code optimization strategies: reducing register pressure, controlling thread granularity, caching data in local/shared memory and common sub-expression elimination.
- Two resource counters: register usage per thread and amount of cached data per thread-block.



Array reversal

First case

$$\begin{cases} 2sB \leq Z_B \\ 4 \leq R_B \end{cases}$$

strategies (1) (4a) (3a) (2) (2) applied

```
meta_schedule cache(a, c) {
  meta_for (int i = 0; i < dim; i++)
    meta_for (int j = 0; j < B; j++)
      for (int k = 0; k < s; ++k) {
        int x = (i*s+k)*B+j;
        int y = N-1-x;
        c[y] = a[x];
      }
}
```

Second case

$$\begin{cases} 2B \leq Z_B < 2sB \\ 3 \leq R_B \end{cases}$$

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

```
meta_schedule cache(a, c) {
  meta_for (int i = 0; i < dim; i++)
    meta_for (int j = 0; j < B; j++) {
      int x = i*B+j;
      int y = N-1-x;
      c[y] = a[x];
    }
}
```

$$\begin{cases} 2B \leq Z_B < 2sB \\ 3 \leq R_B < 4 \end{cases}$$

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

Third case

$$\begin{cases} Z_B < 2B \\ 3 \leq R_B \end{cases}$$

strategies (1) (3b) (2) (2) (4b) applied

```
meta_schedule {
  meta_for (int i = 0; i < dim; i++)
    meta_for (int j = 0; j < B; j++) {
      int x = i*B+j;
      int y = N-1-x;
      c[y] = a[x];
    }
}
```

$$\begin{cases} Z_B < 2B \\ 3 \leq R_B < 4 \end{cases}$$

strategies (2) (2) (3b) (1) (4b) applied

1D Jacobi (1/2)

```
int T, N, s, B, 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;
          }
}
```

- CSE strategy is applied successfully for all cases.
- Post-processing is needed for calculating the total amount of required shared memory per thread-block, due to the fact that array a has multiple accesses and that each access has a different index.

1D Jacobi (2/2)

First case

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

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

```
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

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

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

```
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

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

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

```
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;
      }
  }
```

Matrix matrix multiplication

First case

$$\begin{cases} sB_0B_1 + sBB_1 + B_0B \leq Z_B \\ 9 \leq R_B \end{cases}$$

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

```
meta_schedule cache(a, b, c) {
  meta_for (int v0 = 0; v0 < dim0; v0++)
    meta_for (int v1 = 0; v1 < dim1; v1++)
      for (int w = 0; w < dim; w++)
        meta_for (int u0=0; u0<B0; u0++)
          meta_for (int u1=0; u1<B1; u1++)
            for (int k = 0; k < s; ++k) {
              int i = v0*B0+u0;
              int j = (v1*s+k)*B1+u1;
              for (int z = 0; z < B; z++) {
                int p = B*w+z;
                c[i*N+j] = c[i*N+j] +
                           a[i][p] * b[p][j];
              }
            }
}
```

Second case

$$\begin{cases} B_0B_1 + BB_1 + B_0B \leq Z_B \\ Z_B < sB_0B_1 + sBB_1 + B_0B \\ 8 \leq R_B \end{cases}$$

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

$$\begin{cases} B_0B_1 + BB_1 + B_0B \leq Z_B \\ Z_B < sB_0B_1 + sBB_1 + B_0B \\ 8 \leq R_B < 9 \end{cases}$$

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

```
meta_schedule cache(a, b, c) {
  meta_for (int v0 = 0; v0 < dim0; v0++)
    meta_for (int v1 = 0; v1 < dim1; v1++)
      for (int w = 0; w < dim; w++)
        meta_for (int u0=0; u0<B0; u0++)
          meta_for (int u1=0; u1<B1; u1++)
          {
            int i = v0*B0+u0;
            int j = v1*B1+u1;
            for (int z = 0; z < B; z++) {
              int p = B*w+z;
              c[i*N+j] = c[i*N+j] +
                         a[i][p] * b[p][j];
            }
          }
}
```

Third case

$$\begin{cases} Z_B < B_0B_1 + BB_1 + B_0B \\ 8 \leq R_B \end{cases}$$

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

$$\begin{cases} Z_B < B_0B_1 + BB_1 + B_0B \\ 8 \leq R_B < 9 \end{cases}$$

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

```
meta_schedule {
  meta_for (int v0 = 0; v0 < dim0; v0++)
    meta_for (int v1 = 0; v1 < dim1; v1++)
      for (int w = 0; w < dim; w++)
        meta_for (int u0=0; u0<B0; u0++)
          meta_for (int u1=0; u1<B1; u1++)
          {
            int i = v0*B0+u0;
            int j = v1*B1+u1;
            for (int z = 0; z < B; z++) {
              int p = B*w+z;
              c[i*N+j] = c[i*N+j] +
                         a[i][p] * b[p][j];
            }
          }
}
```

Case selection at runtime (1/2)

- Once compiled and optimized, a parametric `omp target` region is encoded by a sequence of pairs

$$(C_1 K_1), \ldots, (C_e, K_e)$$

  where $C_1, \ldots, C_e$ are systems of algebraic constraints and $K_1, \ldots, K_e$ are parametric CUDA kernels (in PTX).
- At runtime, the hardware parameters $R_1, \ldots, R_s$ and the data parameters $D_1, \ldots, D_u$ are known
- the program parameters $E_1, \ldots, E_v$ remained to be determined.

- By the way those constraint systems are built, testing consistency is trivial
- if C_i is consistent, then the kernel K_i is excluded
- For the consistent C_i's one can determine realistic values of $E_1, \ldots, E_v$ by solving "small" LP problems.
- All those LP problems can be computed by a single kernel launch on the device, with, say each thread-block solving one LP.

Case selection at runtime (2/2)

$$\begin{cases} sB_0B_1 + sBB_1 + B_0B \le Z_B \\ 32 \le B_0B_1 \le 256 \\ 32 \le B^2 \le 256 \\ 1 \le s \end{cases}$$

```
meta_schedule cache(a, b, c) {
  meta_for (int v0 = 0; v0 < dim0; v0++)
    meta_for (int v1 = 0; v1 < dim1; v1++)
      for (int w = 0; w < dim; w++)
        meta_for (int u0=0; u0<B0; u0++)
          meta_for (int u1=0; u1<B1; u1++)
            for (int k = 0; k < s; ++k) {
              int i = v0*B0+u0;
              int j = (v1*s+k)*B1+u1;
              for (int z = 0; z < B; z++) {
                int p = B*w+z;
                c[i*N+j] = c[i*N+j] +
                          a[i][p] * b[p][j];
              }
            }
}
```

$$\begin{cases} 2^x 2^y 2^z + 2^x 2^t 2^z + 2^y 2^t \le Z_B \\ 32 \le 2^{y+z} \le 256 \\ 32 \le 2^{2t} \le 256 \\ 1 \le 2^x \end{cases} \Rightarrow \begin{cases} 3(2^{max(x+y+z,x+t+z,y+t)}) \le Z_B \\ 32 \le 2^{y+z} \le 256 \\ 32 \le 2^{2t} \le 256 \\ 1 \le 2^x \end{cases} \Rightarrow \begin{cases} max(x+y+z+t) \\ x+y+z \le 14 \\ x+t+z \le 14 \\ y+t \le 14 \\ 5 \le y+z \le 8 \\ 5 \le 2t \le 8 \\ 0 \le x \end{cases}$$

With $Z_B = 48kB$, the solution of the above linear programming is $t = 4, x = 2, y = 4, z = 4$, that is,

$$B = 16, s = 4, B_0 = 16, B_1 = 16$$

Plan

Conclusion and future work

- We have shown how, from an annotated C/C++ program, parametric CUDA kernels could be optimized.
- These optimized parametric CUDA kernels are organized in the form of a case discussion, where cases depend on the values of machine parameters (e.g. hardware resource limits) and program parameters (e.g. dimension sizes of thread-blocks).
- Our preliminary implementation uses LLVM, MAPLE and PPCG;
- it successfully processes a variety of standard test-examples. In particular, the computer algebra portion of the computations is not a bottleneck.