

# Comprehensive Optimization of Parametric Kernels for Graphics Processing Units

Xiaohui Chen<sup>1</sup>, Marc Moreno Maza<sup>2</sup>, Jeeva Paudel<sup>3</sup>, Ning Xie<sup>4</sup>

<sup>1</sup> AMD, Markham, Canada

<sup>2</sup> U. Western Ontario, London, Canada

<sup>3</sup> IBM Canada Ltd, Markham, Canada

<sup>4</sup> Huawei Technologies Canada, Markham, Canada

Applications of Computer Algebra

Session on High-Performance Computer Algebra

Jerusalem College of Technology, July 21, 2017

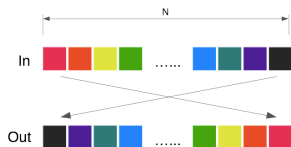
# Automatic CUDA code generation (1/2)

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));

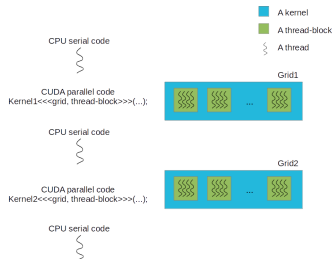
// Copying the data from host to device
cudaMemcpy(dev_In, In, (N)*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_Out, Out, (N)*sizeof(int), cudaMemcpyHostToDevice);

// 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

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);
```

$$C_1: \begin{cases} B \leq Z \\ 6 \leq R < 8 \end{cases}$$

$$C_2: \begin{cases} 2B \leq Z \\ 8 \leq R \end{cases}$$

Z: maximum number of shared memory words per SM.

R: maximum number of registers per thread.



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

# The MetaFork language

- ▶ The MetaFork framework<sup>1</sup> 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

---

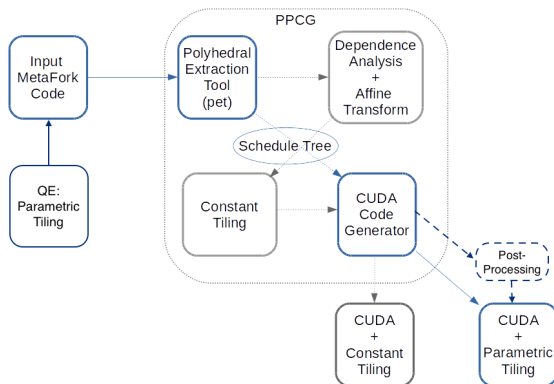
<sup>1</sup>Xiaohui Chen, Marc Moreno Maza, Sushek Shekar, and Priya Unnikrishnan. MetaFork: A framework for concurrency platforms targeting multicores. In *Proceedings of IWOMP'14*, pages 30-44, 2014.

# 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<sup>2</sup> allows the automatic generation of kernels depending on parameters.
- ▶ This MetaFork-to-CUDA code generator modifies and extends PPCG<sup>3</sup>, a C-to-CUDA compilation framework.

<sup>2</sup>Publicly available at <http://www.metafork.org/>

<sup>3</sup>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];  
        }  
}
```

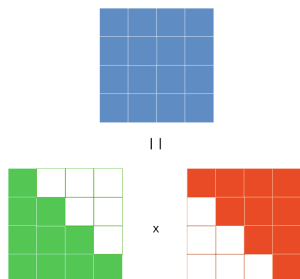
Table: Speedup comparison between PPCG and MetaFork kernel code

Speedup (kernel)	Input size		
	2 <sup>23</sup>	2 <sup>24</sup>	2 <sup>25</sup>
Thread-block size			
	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	<b>20.451</b>	<b>21.614</b>	<b>22.965</b>
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

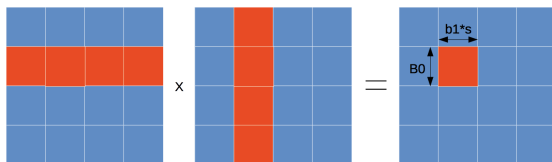
Table: Speedup comparison between PPCG and MetaFork kernel code



Speedup (kernel)				Input size	
Thread-block size				$2^{10} * 2^{10}$	$2^{11} * 2^{11}$
kernel0, kernel1					
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)

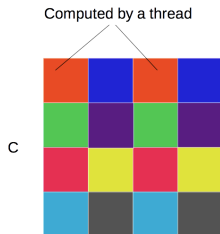
Table: Speedup factors obtained with kernels generated by PPCG and MetaFork with granularity, respectively, w.r.t. the serial C code with good data locality

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



- Generation of parametric CUDA kernels
- 2 Comprehensive Optimization of Parametric Kernels
  - Comprehensive optimization: input and output
  - Experimentation
- 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);
```

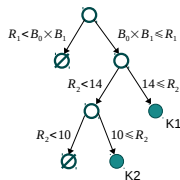
$$C_2: \begin{cases} B_0 \times B_1 \leq R_2 \\ 10 \leq R_1 < 14 \end{cases}$$

```
__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.

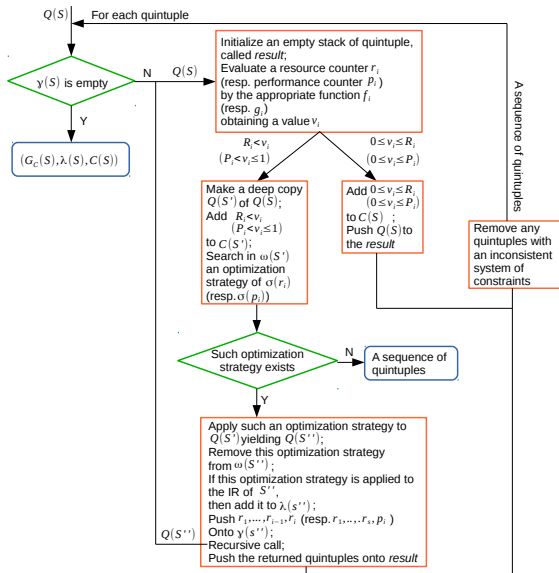
# Comprehensive optimization

- ▶ Consider a **code fragment**  $\mathcal{S}$  written in the C language.
- ▶ Let  $R_1, \dots, R_s$  the **hardware resource limits** of the targeted hardware device.
- ▶ Let  $D_1, \dots, D_u$  and  $E_1, \dots, E_v$  be the data parameters and program parameters of  $\mathcal{S}$  (i.e scalar variable read but not written in  $\mathcal{S}$ )
- ▶ Let  $P_1, \dots, P_t$  be **performance measures** of a program running on the device.
- ▶ Functions to evaluate hardware and performance counters of  $\mathcal{S}$
- ▶ Let  $O_1, \dots, O_w$  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  $C_1, \dots, C_e$  and corresponding programs  $K_1, \dots, K_e$  such that

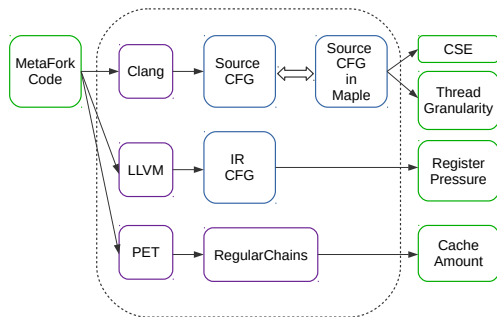
- (i) [**constraint soundness**] each  $C_1, \dots, C_e$  is consistent
- (ii) [**code soundness**] each  $K_i$  is a faithful translation of  $\mathcal{S}$  whenever  $C_i$  holds
- (iii) [**coverage**] for every run of  $\mathcal{S}$ , there exists a  $K_i$  producing the same result (under the conditions 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

Second case

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

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

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

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

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

strategies (2) (2) (3b) (1) (4b) 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];
      }
}
```

```
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];
    }
}
```

```
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];
    }
}
```



# 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];
    }
  }
}
```

- 1 Generation of parametric CUDA kernels
- 2 Comprehensive Optimization of Parametric Kernels
- 3 Conclusion and future work

# 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.