Harnessing CUDA Dynamic Parallelism for the Solution of Sparse Linear Systems

José I. Aliaga[1] Davor Davidović[2] Joaquín Pérez[1]
Enrique S. Quintana-Ortí[1]

[1]Departamento de Ingeniería y Ciencia de los Computadores, Universitat Jaume I, 12.071–Castellón de la Plana, (Spain)
{aliaga,joaquin.perez,quintana}@uji.es

[2]Institut Ruđer Bošković, Centar za Informatiku i Računarstvo - CIR, Zagreb (Croatia)
davor.davidovic@irb.hr

September, 2015

Motivation and Introduction

- The discretization of PDE's often leads to linear systems

$$Ax = b$$

where

  - $A \in \mathbb{R}^{n,n}$ is a **large** and **sparse** coefficient matrix
  - $x \in \mathbb{R}^n$ is the sought-after solution
  - $b \in \mathbb{R}^n$ is a given r.h.s. vector

- Direct or iterative methods can be used to solve them
  - For 3D problems, iterative methods based on Krylov subspaces
  - The structure and numerical kernels are similar in all these ones
  - For the SPD case, the Conjugate Gradient (CG) should be applied

Motivation and Introduction

- The discretization of PDE's often leads to linear systems

$$Ax = b$$

where

  - $A \in \mathbb{R}^{n,n}$ is a **large** and **sparse** coefficient matrix
  - $x \in \mathbb{R}^n$ is the sought-after solution
  - $b \in \mathbb{R}^n$ is a given r.h.s. vector

- Direct or iterative methods can be used to solve them
  - For 3D problems, iterative methods based on Krylov subspaces
  - The structure and numerical kernels are similar in all these ones
  - For the SPD case, the Conjugate Gradient (CG) should be applied

Motivation and Introduction

Conjugate Gradient Algorithm

Initialize $r_0, p_0, x_0, \sigma_0, \tau_0; j := 0$	
while $(\tau_j > \tau_{\max})$	Loop for iterative CG solver
$v_j := Ap_j$	O1. SPMV
$\alpha_j := \sigma_j / p_j^T v_j$	O2. DOT
$x_{j+1} := x_j + \alpha_j p_j$	O3. AXPY
$r_{j+1} := r_j - \alpha_j v_j$	O4. AXPY
$\zeta_j := r_{j+1}^T r_{j+1}$	O5. DOT
$\beta_j := \zeta_j / \sigma_j$	O6. Scalar op
$\sigma_{j+1} := \zeta_j$	O7. Scalar op
$p_{j+1} := r_{j+1} + \beta_j p_j$	O8. XPAY (AXPY-like)
$\tau_{j+1} := \| r_{j+1} \|_2 = \sqrt{\zeta_j}$	O9. Vector 2-norm (in practice, sqrt)
$j := j + 1$	
endwhile	

Motivation and Introduction

- When we use a heterogeneous server (CPU + GPU)
  - The computations of CG rely on the GPU
  - A CPU thread is in charge of controlling the GPU
- Aspects to consider in GPU computation,
  - The data communication via the slow PCI-e bus blurs the computational cost
  - The invocation of fine-grain kernels prevents the CPU from entering an energy-efficient C-state, increasing the energy consumption

Motivation and Introduction

- When we use a heterogeneous server (CPU + GPU)
  - The computations of CG rely on the GPU
  - A CPU thread is in charge of controlling the GPU
- Aspects to consider in GPU computation,
  - The data communication via the slow PCI-e bus blurs the computational cost
  - The invocation of fine-grain kernels prevents the CPU from entering an energy-efficient C-state, increasing the energy consumption

Motivation and Introduction

Conjugate Gradient Algorithm

Initialize $r_0, p_0, x_0, \sigma_0, \tau_0; j := 0$	
while $(\tau_j > \tau_{\max})$	cuBLAS/cuSPARSE routines
$v_j := Ap_j$	`cusparseScsrmv`
$\alpha_j := \sigma_j / p_j^T v_j$	`cublasSdot` + Scalar op
$x_{j+1} := x_j + \alpha_j p_j$	`cublasSaxpy`
$r_{j+1} := r_j - \alpha_j v_j$	`cublasSaxpy`
$\zeta_j := r_{j+1}^T r_{j+1}$	`cublasSdot`
$\beta_j := \zeta_j / \sigma_j$	Scalar op
$\sigma_{j+1} := \zeta_j$	Scalar op
$p_{j+1} := r_{j+1} + \beta_j p_j$	`cublasSscal+cublasSaxpy`
$\tau_{j+1} :=\| r_{j+1} \|_2 = \sqrt{\zeta_j}$	Vector 2-norm (in practice, sqrt)
$j := j + 1$	
endwhile	

Motivation and Introduction

- When we use a heterogeneous server (CPU + GPU)
  - The computations of CG rely on the GPU
  - A CPU thread is in charge of controlling the GPU
- Aspects to consider in GPU computation,
  - The data communication via the slow PCI-e bus blurs the computational cost
    $\rightarrow$ Store all data on the GPU memory
  - The invocation of fine-grain kernels prevents the CPU from entering an energy-efficient C-state, increasing the energy consumption

Motivation and Introduction

- When we use a heterogeneous server (CPU + GPU)
  - The computations of CG rely on the GPU
  - A CPU thread is in charge of controlling the GPU
- Aspects to consider in GPU computation,
  - The data communication via the slow PCI-e bus blurs the computational cost
    $\rightarrow$ Store all data on the GPU memory
  - The invocation of fine-grain kernels prevents the CPU from entering an energy-efficient C-state, increasing the energy consumption
    $\rightarrow$ Reduce the number of kernels
    $\rightarrow$ Grow the grain of the kernels

Motivation and Introduction

- The first solution was to fuse CUDA kernels
  - To develop specific CUDA kernels to solve any operation
  - Only a transfer to verify the loop condition
  - To merge CUDA kernels

  Systematic Fusion of CUDA Kernels for Iterative Sparse Linear System Solvers (EUROPAR'15)

- Dynamic Parallelism (DP) is an alternative technique
  - A parent CUDA kernel can launch other child CUDA kernels
  - For the CG case: the CPU only launches a simple CUDA kernel, which is in charge of launch other CUDA kernels
  - The CPU is completely idle during all the CG computations
  - The simple CUDA kernel can include fine-grain or merged kernels

- **Objective**: Development of the DP version of CG

Motivation and Introduction

- The first solution was to fuse CUDA kernels
  - To develop specific CUDA kernels to solve any operation
  - Only a transfer to verify the loop condition
  - To merge CUDA kernels

  Systematic Fusion of CUDA Kernels for Iterative Sparse Linear System Solvers (EUROPAR'15)

- Dynamic Parallelism (DP) is an alternative technique
  - A parent CUDA kernel can launch other child CUDA kernels
  - For the CG case: the CPU only launches a simple CUDA kernel, which is in charge of launch other CUDA kernels
  - The CPU is completely idle during all the CG computations
  - The simple CUDA kernel can include fine-grain or merged kernels

- **Objective**: Development of the DP version of CG

Motivation and Introduction

- The first solution was to fuse CUDA kernels
  - To develop specific CUDA kernels to solve any operation
  - Only a transfer to verify the loop condition
  - To merge CUDA kernels

  Systematic Fusion of CUDA Kernels for Iterative Sparse Linear System Solvers (EUROPAR'15)

- Dynamic Parallelism (DP) is an alternative technique
  - A parent CUDA kernel can launch other child CUDA kernels
  - For the CG case: the CPU only launches a simple CUDA kernel, which is in charge of launch other CUDA kernels
  - The CPU is completely idle during all the CG computations
  - The simple CUDA kernel can include fine-grain or merged kernels

- **Objective**: Development of the DP version of CG

Motivation and Introduction

- The first solution was to fuse CUDA kernels
  - To develop specific CUDA kernels to solve any operation
  - Only a transfer to verify the loop condition
  - To merge CUDA kernels

  Systematic Fusion of CUDA Kernels for Iterative Sparse Linear System Solvers (EUROPAR'15)

- Dynamic Parallelism (DP) is an alternative technique
  - A parent CUDA kernel can launch other child CUDA kernels
  - For the CG case: the CPU only launches a simple CUDA kernel, which is in charge of launch other CUDA kernels
  - The CPU is completely idle during all the CG computations
  - The simple CUDA kernel can include fine-grain or merged kernels

- **Objective**: Development of the DP version of CG

Outline

Outline

CUDA kernels of CG

- Each CG operation can be implemented as a CUDA kernel
- The scalar operations
- The vector operations
  - The kernels of truly parallel operations are easily developed
    $\rightarrow$ axpy, axpy-like, scal, ...
  - Other operations usually require a collaboration between threads
    $\rightarrow$ dot, ...
- For SPMV,
  - CSR format: spmv_csr_scalar_kernel, spmv_csr_vector_kernel
  - ELLPACK format: spmv_ell_kernel

CUDA kernels of CG

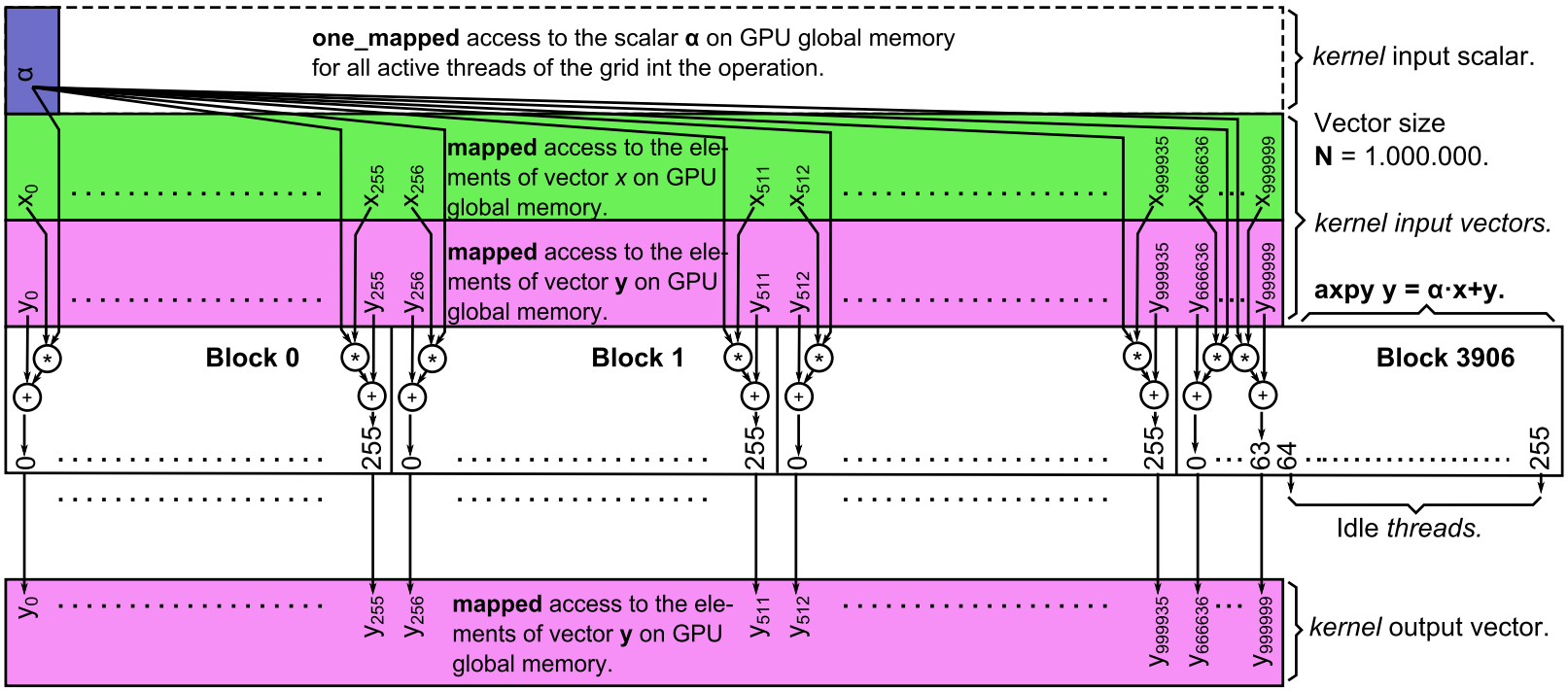
- Each CG operation can be implemented as a CUDA kernel
- The scalar operations can be made by a thread of a block
- The vector operations
  - The kernels of truly parallel operations are easily developed
    $\rightarrow$ axpy, axpy-like, scal, ...
  - Other operations usually require a collaboration between threads
    $\rightarrow$ dot, ...
- For SPMV,
  - CSR format: spmv_csr_scalar_kernel, spmv_csr_vector_kernel
  - ELLPACK format: spmv_ell_kernel

CUDA kernels of CG

- Each CG operation can be implemented as a CUDA kernel
- The scalar operations can be made by a thread of a block
- The vector operations compute on complex grids ($Bs = 256$)
  - The kernels of truly parallel operations are easily developed
    $\rightarrow$ axpy, axpy-like, scal, ...
  - Other operations usually require a collaboration between threads
    $\rightarrow$ dot, ...
- For SPMV,
  - CSR format: spmv_csr_scalar_kernel, spmv_csr_vector_kernel
  - ELLPACK format: spmv_ell_kernel

cudaSaxpy

```
1  __global__ void cudaSaxpy (int n, float *alpha, float *x, float *y) {
2    unsigned int BlkSize = blockDim.x;                    // = 256
3    unsigned int tid     = threadIdx.x;
4    unsigned int i       = blockIdx.x * BlkSize + tid;
5
6    if (i < n) {
7      y[i] +=  *alfa * x[i];
8    }
9  }
```



CUDA kernels of CG

- Each CG operation can be implemented as a CUDA kernel
- The scalar operations can be made by a thread of a block
- The vector operations compute on complex grids ($Bs = 256$)
  - The kernels of truly parallel operations are easily developed
    $\rightarrow$ axpy, axpy-like, scal, ...
  - Other operations usually require a collaboration between threads
    $\rightarrow$ dot, ...
- For SPMV,
  - CSR format: spmv_csr_scalar_kernel, spmv_csr_vector_kernel
  - ELLPACK format: spmv_ell_kernel

cudaSdot

- The DOT can be decomposed into two operations:
  - Element-wise product (EWP): ($z = x .* y$)
  - Reduction of the result (addition of its elements): ($\alpha = \sum z$)
- An iterative process has to be implemented to reduce the vector
  - The kernels have an input vector (*in*) and an output vector (*out*)
  - One element of *ini* is assigned to a single thread
  - Each block computes the addition of the local values of *in*, storing the result in the position of *out* related to the block id
  - The ratio of the sizes of the two vectors is equal to the block size
  - The process ends when the size of *out* is equal to 1

cudaSdot

- The DOT can be decomposed into two operations:
  - Element-wise product (EWP): ($z = x. * y$)
  - Reduction of the result (addition of its elements): ($\alpha = \sum z$)
- An iterative process has to be implemented to reduce the vector
  - The kernels have an input vector (*in*) and an output vector (*out*)
  - One element of *ini* is assigned to a single thread
  - Each block computes the addition of the local values of *in*, storing the result in the position of *out* related to the block id
    - The use of shared memory accelerates the computation
    - Block level synchronizations are required to avoid errors
  - The ratio of the sizes of the two vectors is equal to the block size
  - The process ends when the size of *out* is equal to 1

cudaSdot

```
1  __global__ void cudaSreduce (int n, float *in, float *out) {
2    extern __shared__ float vtmp[];
3
4    // Each thread loads one element from global to shared mem
5    unsigned int BlkSize = blockDim.x;                  // = 256
6    unsigned int tid     = threadIdx.x;                 // block thread index
7    unsigned int i       = blockIdx.x * BlkSize + tid;
8
9    vtmp[tid] = ( i < n ) ? in[i] : 0; __syncthreads();
10
11   // Reduce from BlkSize=256 elements to 128, 64, 32, 16, 8, 2 and 1
12   if (tid < 128) { vtmp[tid] += vtmp[tid + 128]; } __syncthreads();
13   if (tid < 64)  { vtmp[tid] += vtmp[tid + 64 ]; } __syncthreads();
14   if (tid < 32)  {
15     volatile float *vtmp2 = vtmp;
16     vtmp2[tid] += vtmp2[tid + 32]; vtmp2[tid] += vtmp2[tid + 16];
17     vtmp2[tid] += vtmp2[tid + 8 ]; vtmp2[tid] += vtmp2[tid + 4 ];
18     vtmp2[tid] += vtmp2[tid + 2 ]; vtmp2[tid] += vtmp2[tid + 1 ];
19   }
20
21   // Write result for this block to global mem
22   if (tid == 0) out[blockIdx.x] = vtmp[0];
23 }
```

cudaSdot

- The DOT can be decomposed into two operations:
  - Element-wise product (EWP): ($z = x .* y$)
  - Reduction of the result (addition of its elements): ($\alpha = \sum z$)
- An iterative process has to be implemented to reduce the vector
  - The kernels have an input vector (*in*) and an output vector (*out*)
  - One element of *ini* is assigned to a single thread
  - Each block computes the addition of the local values of *in*, storing the result in the position of *out* related to the block id
    - The use of shared memory accelerates the computation
    - Block level synchronizations are required to avoid errors
  - The ratio of the sizes of the two vectors is equal to the block size
  - The process ends when the size of *out* is equal to 1

cudaSdot

```
1  void cudaSreduction ( int Gs, int Bs, int Ms,
2                        int n, float z[],
3                        float vtmp[] , float vtmp2[] ) {
4
5    float *aux1 = vtmp, *aux2 = vtmp2;
6    int b = 1, Gs_next;
7
8
9
10   // reduce blocks
11   cudaSreduce <<< Gs, Bs, Ms >>> ( n,  z  , aux1 );
12
13   while( Gs > 1 ){
14     Gs_next = ( unsigned int ) ceil( ( float ) Gs / Bs );
15     // reduce blocks
16     cudaSreduce <<< Gs_next, Bs, Ms >>> ( Gs, aux1, aux2 );
17     Gs = Gs_next;
18     b = 1 - b;
19     if( b ){ aux1 = vtmp; aux2 = vtmp2; }
20     else { aux2 = vtmp; aux1 = vtmp2; }
21   }
22
23   // Write result
24   if( b == 0 )
25     cudaMemcpy( vtmp, aux1, sizeof( float ), cudaMemcpyDeviceToDevice );
26 }
```

cudaSdot

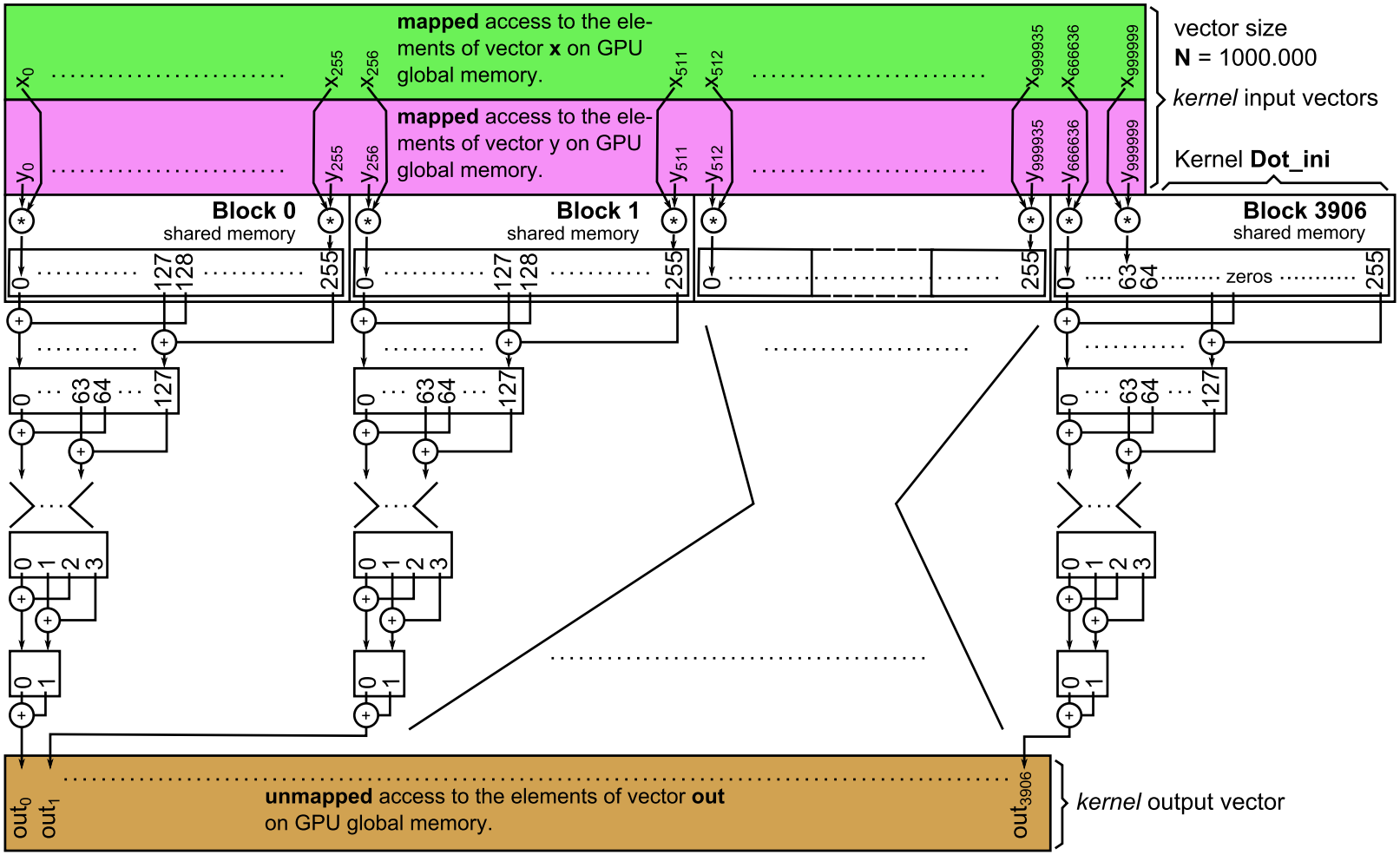
```
1  void cudaSdot        ( int Gs, int Bs, int Ms,
2                         int n, float x[], float y[],
3                         float vtmp[] , float vtmp2[] ) {
4
5    float *aux1 = vtmp, *aux2 = vtmp2;
6    int b = 1, Gs_next;
7
8    // element-wise product
9    cudaSewp (n, alpha, x, y, aux2);
10   // reduce blocks
11   cudaSreduce <<< Gs, Bs, Ms >>> ( n, aux2, aux1 );
12
13   while( Gs > 1 ){
14     Gs_next = ( unsigned int ) ceil( ( float ) Gs / Bs );
15     // reduce blocks
16     cudaSreduce <<< Gs_next, Bs, Ms >>> ( Gs, aux1, aux2 );
17     Gs = Gs_next;
18     b = 1 - b;
19     if( b ){ aux1 = vtmp; aux2 = vtmp2; }
20     else { aux2 = vtmp; aux1 = vtmp2; }
21   }
22
23   // Write result
24   if( b == 0 )
25     cudaMemcpy( vtmp, aux1, sizeof( float ), cudaMemcpyDeviceToDevice );
26 }
```

cudaSdot

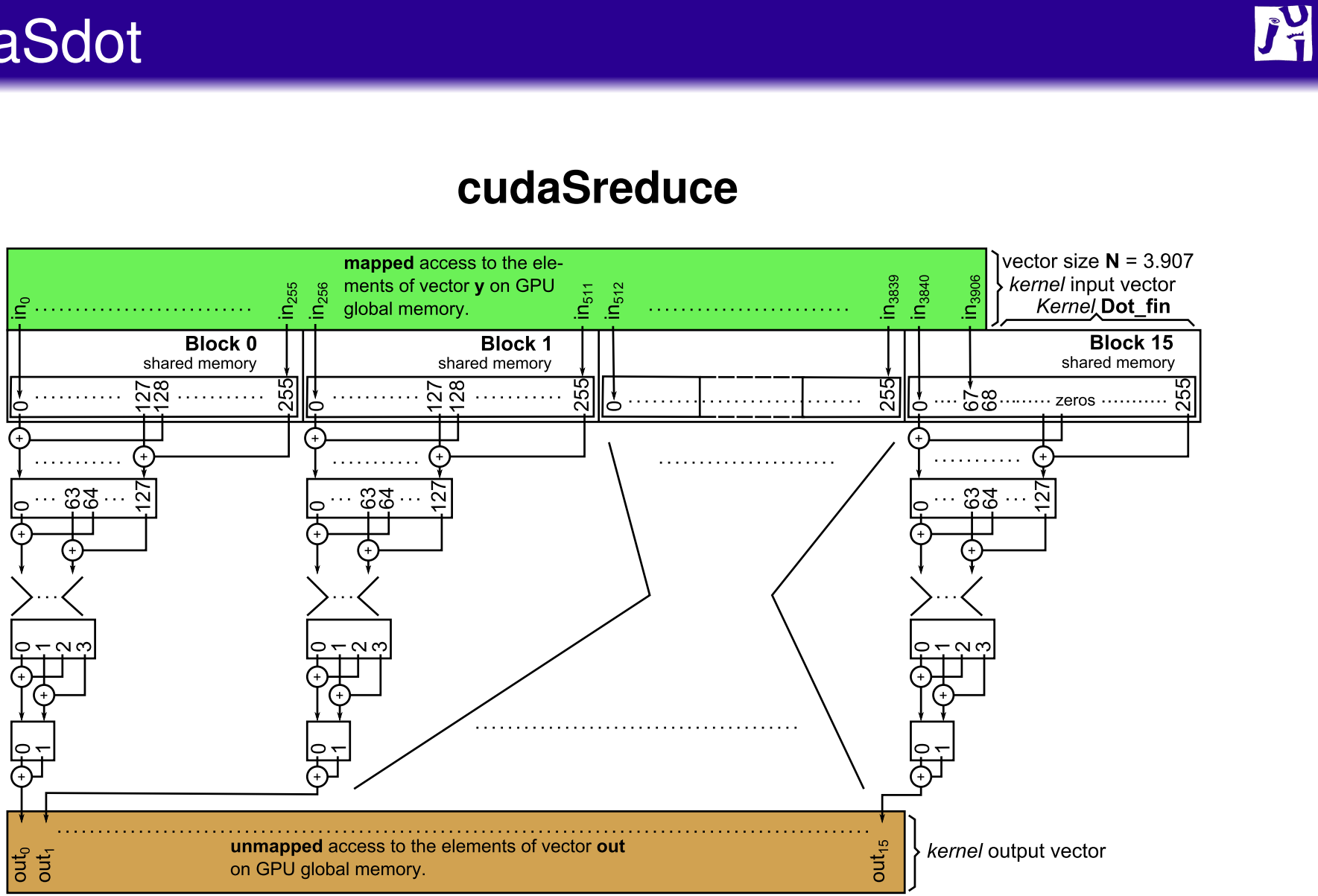
```
1  void cudaSdot        ( int Gs, int Bs, int Ms,
2                         int n, float x[], float y[],
3                         float vtmp[] , float vtmp2[] ) {
4
5    float *aux1 = vtmp, *aux2 = vtmp2;
6    int b = 1, Gs_next;
7
8
9    // element-wise product and reduce blocks
10   cudaSewp_reduce (n, alpha, x, y, aux1);
11
12
13   while( Gs > 1 ){
14     Gs_next = ( unsigned int ) ceil( ( float ) Gs / Bs );
15     // reduce blocks
16     cudaSreduce <<< Gs_next, Bs, Ms >>> ( Gs, aux1, aux2 );
17     Gs = Gs_next;
18     b = 1 - b;
19     if( b ){ aux1 = vtmp; aux2 = vtmp2; }
20     else { aux2 = vtmp; aux1 = vtmp2; }
21   }
22
23   // Write result
24   if( b == 0 )
25     cudaMemcpy( vtmp, aux1, sizeof( float ), cudaMemcpyDeviceToDevice );
26 }
```

cudaSdot

cudaSewp_reduce

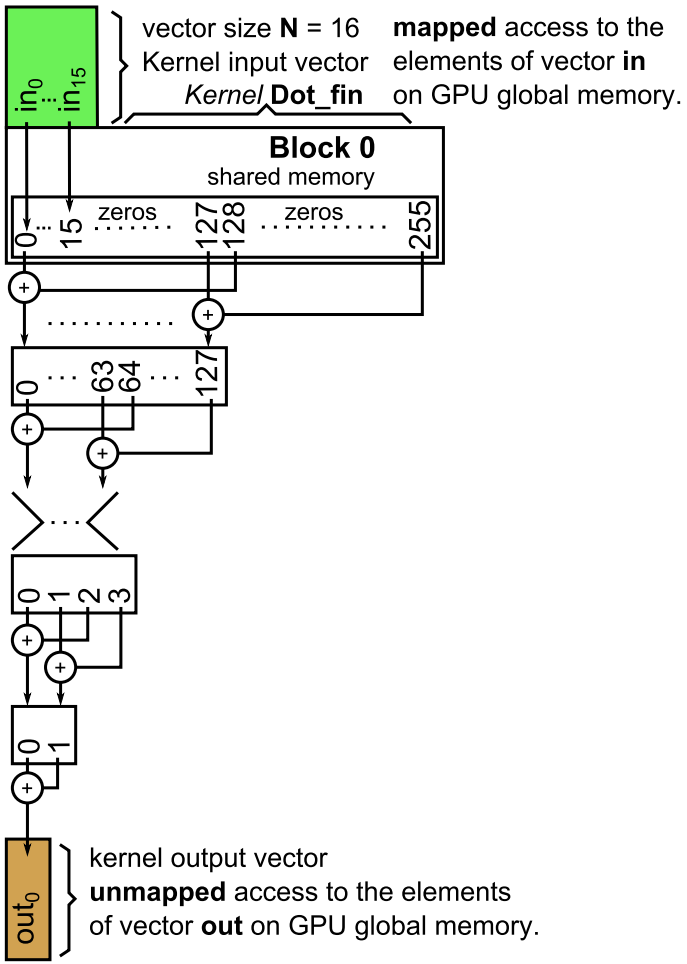


cudaSdot



cudaSreduce

cudaSdot

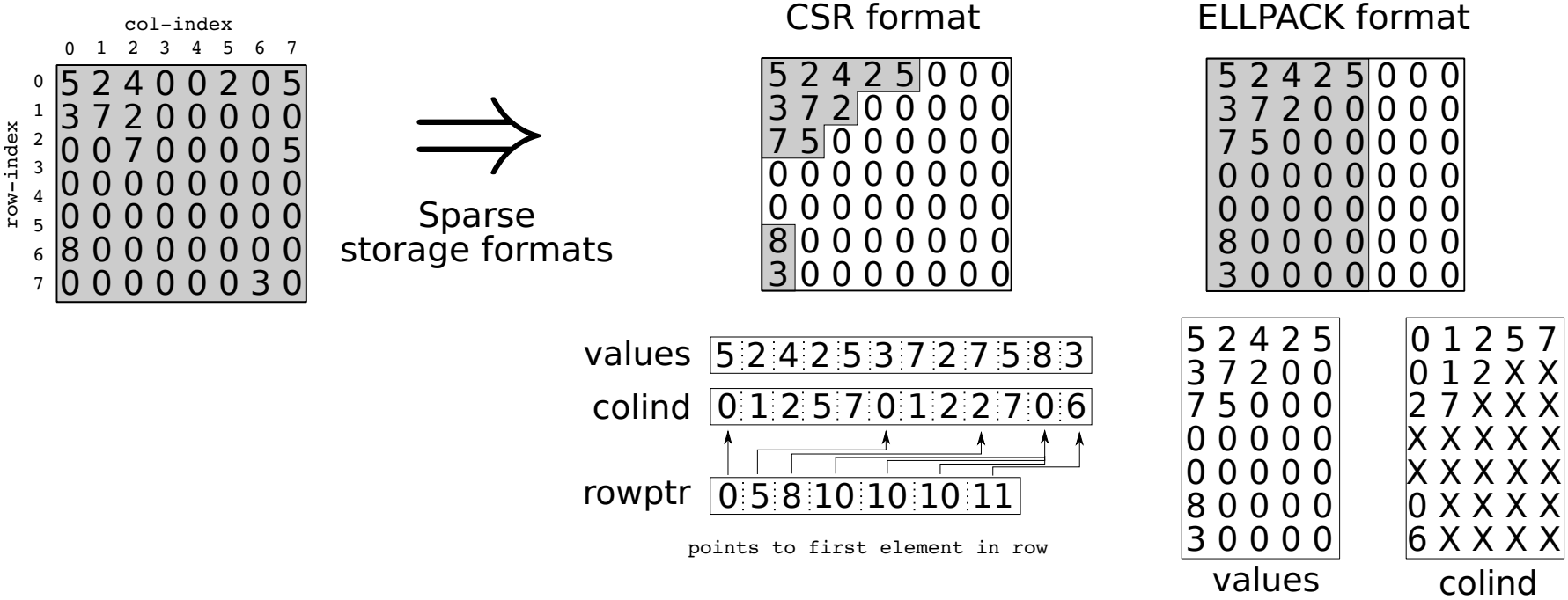

cudaSreduce
vector size N = 16
Kernel input vector
Kernel Dot_fin
mapped access to the
elements of vector in
on GPU global memory.
in0
in15
Block 0
shared memory
zeros
zeros
0
15
127
128
255
63
64
1
2
3
out0
kernel output vector
unmapped access to the elements
of vector out on GPU global memory.

CUDA kernels of CG

- Each CG operation can be implemented as a CUDA kernel
- The scalar operations can be made by a thread of a block
- The vector operations compute on complex grids ($Bs = 256$)
  - The kernels of truly parallel operations are easily developed $\rightarrow$ axpy, axpy-like, scal, ...
  - Other operations usually require a collaboration between threads $\rightarrow$ dot, ...
- For SPMV, we use three kernels defined by N. Bell and M. Garland in NVR-2008-004
  - CSR format: spmv_csr_scalar_kernel, spmv_csr_vector_kernel
  - ELLPACK format: spmv_ell_kernel

cudaSspmv

Sparse Matrix Formats



cudaSspmv

- spmv_csr_scalar_kernel
  - A thread of a block computes one element of the result
  - Each thread works without any collaboration
  - The maximum dimension for a 1D Grid is

$$65,535 * Bs = 65,535 * 256 = 16,776,960$$

- spmv_csr_vector_kernel
  - Each element of the result is computed by a warp (32 threads)
  - The final value requires a reduction step into a warp
  - The maximum dimension for a 1D Grid is

$$65,535 * Bs/32 = 65,535 * 256/32 = 16,776,960/32 = 524,280$$

- spmv_ell_kernel
  - The same features as spmv_csr_scalar_kernel
  - The coalescent data access reduces the execution time

cudaSspmv

- spmv_csr_scalar_kernel
  - A thread of a block computes one element of the result
  - Each thread works without any collaboration
  - The maximum dimension for a 1D Grid is

$$65,535 * Bs = 65,535 * 256 = 16,776,960$$

- spmv_csr_vector_kernel
  - Each element of the result is computed by a warp (32 threads)
  - The final value requires a reduction step into a warp
  - The maximum dimension for a 1D Grid is

$$65,535 * Bs/32 = 65,535 * 256/32 = 16,776,960/32 = 524,280$$

- spmv_ell_kernel
  - The same features as spmv_csr_scalar_kernel
  - The coalescent data access reduces the execution time

cudaSspmv

```
1  __global__ void spmv_csr_vector_kernel    ( int num_rows, int *ptr, int *indices,
2                                              float *data, float *x, float *y ) {
3    __shared__ float vals[];                                  // shared memory values
4    int tid       = threadIdx.x;                              // block thread index
5    int BlkSize   = blockDim.x;                               // = 256
6
7    int thread_id = blockIdx.x * BlkSize + tid;               // global thread index
8    int warp_id   = thread_id / 32;                           // global warp index
9    int lane      = thread_id & (32 - 1);                     // thread index within
       the warp
10
11   // one warp per row
12   int row = warp_id;
13
14   if (row < num_rows){
15     int row_start = ptr[row], row_end = ptr[row+1];
16
17     // compute running sum per thread
18     vals[tid] = 0;
19     for(int jj = row_start + lane; jj < row_end; jj += 32)
20       vals[tid] += data[jj] * x[indices[jj]];
21
22     // parallel reduction in shared memory
23     if (lane < 16) vals[tid] += vals[tid+16];
24     if (lane < 8) vals[tid] += vals[tid+8]; if (lane < 4) vals[tid] += vals[tid+4];
25     if (lane < 2) vals[tid] += vals[tid+2]; if (lane < 1) vals[tid] += vals[tid+1];
26
27     // first thread writes the result
28     if (lane == 0) y[row] += vals[tid];
29   }
```

cudaSspmv

```
1  __global__ void spmv_csr_vector_kernel_2D ( int num_rows, int *ptr, int *indices,
2                                               float *data, float *x, float *y ) {
3    __shared__ float vals[];                                    // shared memory values
4    int tid       = threadIdx.x;                                // block thread index
5    int colG      = blockIdx.x * blockDim.x + tid;
6    int rowG      = blockIdx.y * blockDim.y;
7    int thread_id = ( blockDim.x * gridDim.x * rowG ) + colG ; // global thread index
8    int warp_id   = thread_id / 32;                             // global warp index
9    int lane      = thread_id & (32 - 1);                       // thread index within
       the warp
10
11   // one warp per row
12   int row = warp_id;
13
14   if (row < num_rows){
15     int row_start = ptr[row], row_end = ptr[row+1];
16
17     // compute running sum per thread
18     vals[tid] = 0;
19     for(int jj = row_start + lane; jj < row_end; jj += 32)
20       vals[tid] += data[jj] * x[indices[jj]];
21
22     // parallel reduction in shared memory
23     if (lane < 16) vals[tid] += vals[tid+16];
24     if (lane < 8) vals[tid] += vals[tid+8]; if (lane < 4) vals[tid] += vals[tid+4];
25     if (lane < 2) vals[tid] += vals[tid+2]; if (lane < 1) vals[tid] += vals[tid+1];
26
27     // first thread writes the result
28     if (lane == 0) y[row] += vals[tid];
29   }
```

cudaSspmv

- spmv_csr_scalar_kernel
  - A thread of a block computes one element of the result
  - Each thread works without any collaboration
  - The maximum dimension for a 1D Grid is

$$65,535 * Bs = 65,535 * 256 = 16,776,960$$

- spmv_csr_vector_kernel
  - Each element of the result is computed by a warp (32 threads)
  - The final value requires a reduction step into a warp
  - The maximum dimension for a 1D Grid is

$$65,535 * Bs/32 = 65,535 * 256/32 = 16,776,960/32 = 524,280$$

- spmv_ell_kernel
  - The same features as spmv_csr_scalar_kernel
  - The coalescent data access reduces the execution time

Methodology to merge CUDA kernels

- Independent CUDA kernels can always be merged
- Two CUDA kernels related by a RAW dependency,

$$K_1 \xrightarrow{v} K_2, v \in \Re^n$$

can be merged if

  - Both kernels apply the same mapping of threads to the elements of v shared (exchanged) via register
  - Both kernels apply the same mapping of threads blocks to the vector elements shared (exchanged) via shared memory
  - A global barrier is not necessary between the two kernels

- If the kernels K_1 and K_2 can be merged, the grid definition of both kernels should be adjusted

Methodology to merge CUDA kernels

```
1  __global__ void cudaSaxpy ( int n, float *alpha, float *x, float *y) {
2    unsigned int BlkSize = blockDim.x;
3    unsigned int i       = blockIdx.x * BlkSize + threadIdx.x;
4
5    if (i < n) {
6      y[i] +=  *alfa * x[i];
7    }
8  }
9
10 __global__ void cudaSaxpy_1 ( int n, float *alpha1, float *x, float *y,
11                                float *alpha2, float *z) {
12   unsigned int BlkSize = blockDim.x;
13   unsigned int i       = blockIdx.x * BlkSize + threadIdx.x;
14
15   if (i < n) {
16     y[i] +=  *alfa1 * x[i];
17     z[i] +=  *alfa2 * x[i];
18   }
19 }
20
21 // INDEPENDENT KERNELS
22 cudaSaxpy << Gs, Bs >> (n, &alpha1, x, y); // y = y + alpha1 * x
23 cudaSaxpy << Gs, Bs >> (n, &alpha2, x, z); // z = z + alpha2 * x
24
25 // NEW KERNEL
26 cudaSaxpy_1 << Gs, Bs >> (n, &alpha1, &alpha2, x, y, z);
```

Methodology to merge CUDA kernels

- Independent CUDA kernels can always be merged
- Two CUDA kernels related by a RAW dependency,

$$K_1 \xrightarrow{v} K_2, v \in \Re^n$$

  can be merged if
  - Both kernels apply the same mapping of threads to the elements of v shared (exchanged) via register
  - Both kernels apply the same mapping of threads blocks to the vector elements shared (exchanged) via shared memory
  - A global barrier is not necessary between the two kernels
- If the kernels K_1 and K_2 can be merged, the grid definition of both kernels should be adjusted

Methodology to merge CUDA kernels

```
1  __global__ void cudaSaxpy ( int n, float *alpha, float *x, float *y) {
2    unsigned int BlkSize = blockDim.x;
3    unsigned int i       = blockIdx.x * BlkSize + threadIdx.x;
4
5    if (i < n) {
6      y[i] +=  *alfa * x[i];
7    }
8  }
9
10 __global__ void cudaSaxpy_2 ( int n, float *alpha1, float *x, float *y,
11                                float *alpha2, float *z) {
12   unsigned int BlkSize = blockDim.x;
13   unsigned int i       = blockIdx.x * BlkSize + threadIdx.x;
14
15   if (i < n) {
16     y[i] +=  *alfa1 * x[i];
17     z[i] +=  *alfa2 * y[i];
18   }
19 }
20
21 // DEPENDENT KERNELS
22 cudaSaxpy << Gs, Bs >> (n, &alpha1, x, y); // y = y + alpha1 * x
23 cudaSaxpy << Gs, Bs >> (n, &alpha2, y, z); // z = z + alpha2 * y
24
25 // NEW KERNEL
26 cudaSaxpy_2 << Gs, Bs >> (n, &alpha1, &alpha2, x, y, z);
```

Methodology to merge CUDA kernels

```
1  __global__ void cudaSewp_reduce (int n, float *x, float *y, float *out) {
2    extern __shared__ float vtmp[];
3
4    // Each thread loads one element from global to shared mem
5    unsigned int BlkSize = blockDim.x;                  // = 256
6    unsigned int tid     = threadIdx.x;                 // block thread index
7    unsigned int i       = blockIdx.x * BlkSize + tid;
8
9    vtmp[tid] = ( i < n ) ? x[i] * y[i] : 0; __syncthreads();
10
11   // Reduce from BlkSize=256 elements to 128, 64, 32, 16, 8, 2 and 1
12   if (tid < 128) { vtmp[tid] += vtmp[tid + 128]; } __syncthreads();
13   if (tid < 64)  { vtmp[tid] += vtmp[tid + 64 ]; } __syncthreads();
14   if (tid < 32)  {
15     volatile float *vtmp2 = vtmp;
16     vtmp2[tid] += vtmp2[tid + 32]; vtmp2[tid] += vtmp2[tid + 16];
17     vtmp2[tid] += vtmp2[tid + 8 ]; vtmp2[tid] += vtmp2[tid + 4 ];
18     vtmp2[tid] += vtmp2[tid + 2 ]; vtmp2[tid] += vtmp2[tid + 1 ];
19   }
20
21   // Write result for this block to global mem
22   if (tid == 0) out[blockIdx.x] = vtmp[0];
23 }
```

Methodology to merge CUDA kernels

- Independent CUDA kernels can always be merged
- Two CUDA kernels related by a RAW dependency,

$$K_1 \xrightarrow{v} K_2, v \in \Re^n$$

  can be merged if
  - Both kernels apply the same mapping of threads to the elements of v shared (exchanged) via register
  - Both kernels apply the same mapping of threads blocks to the vector elements shared (exchanged) via shared memory
  - A global barrier is not necessary between the two kernels
- If the kernels K_1 and K_2 can be merged, the grid definition of both kernels should be adjusted

Conjugate Gradient Algorithm

Initialize $r_0, p_0, x_0, \sigma_0, \tau_0; j := 0$	
while $(\tau_j > \tau_{\max})$	Loop for iterative CG solver
$v_j := Ap_j$	O1. SPMV
$\alpha_j := \sigma_j / p_j^T v_j$	O2. DOT
$x_{j+1} := x_j + \alpha_j p_j$	O3. AXPY
$r_{j+1} := r_j - \alpha_j v_j$	O4. AXPY
$\zeta_j := r_{j+1}^T r_{j+1}$	O5. DOT
$\beta_j := \zeta_j / \sigma_j$	O6. Scalar op
$\sigma_{j+1} := \zeta_j$	O7. Scalar op
$p_{j+1} := r_{j+1} + \beta_j p_j$	O8. XPAY (AXPY-like)
$\tau_{j+1} := \| r_{j+1} \|_2 = \sqrt{\zeta_j}$	O9. Vector 2-norm (in practice, sqrt)
$j := j + 1$	
endwhile	

Dependency Graph of CG

Initialize $r_0, p_0, x_0, \sigma_0, \tau_0; j := 0$

while $(\tau_j > \tau_{\max})$

O1. SPMV

O2. DOT

O3. AXPY

O4. AXPY

O5. DOTproduct

O6. Scalar op

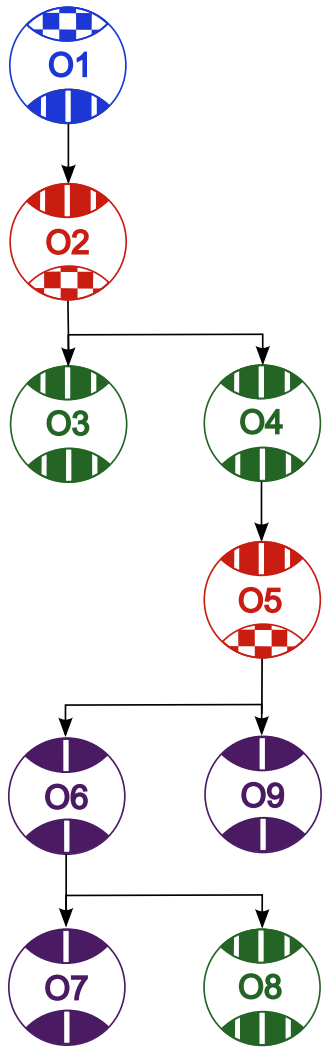
O7. Scalar op

O8. XPAY (AXPY-like)

O9. Vector 2-norm (sqrt)

$j := j + 1$

endwhile



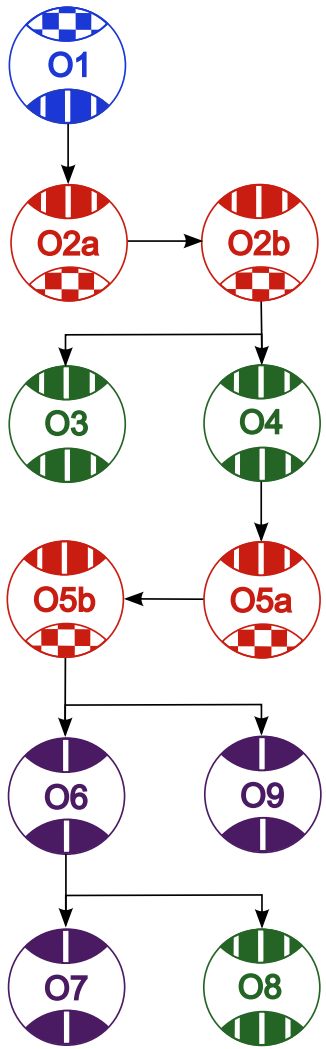
Dependendy Graph of CG for CUDA kernels

Initialize $r_0, p_0, x_0, \sigma_0, \tau_0; j := 0$

while $(\tau_j > \tau_{\max})$

- O1. SPMV
- O2. DOT
- O3. AXPY
- O4. AXPY
- O5. DOTproduct
- O6. Scalar op
- O7. Scalar op
- O8. XPAY (AXPY-like)
- O9. Vector 2-norm (sqrt)
- $j := j + 1$

endwhile



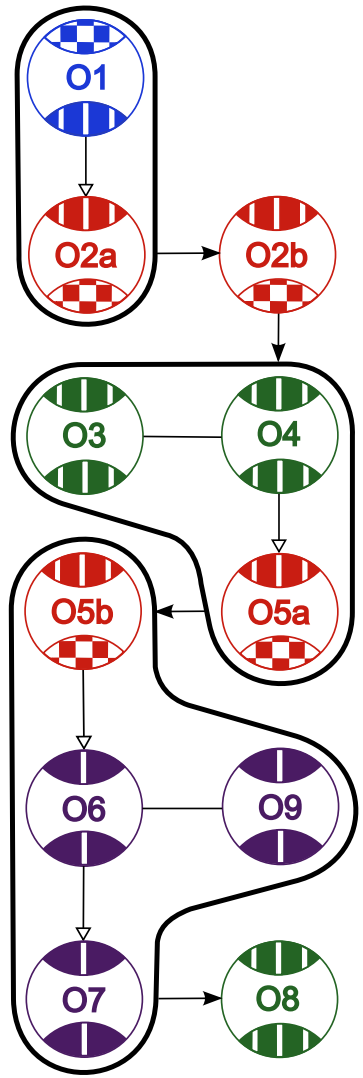
Fusion of CUDA kernels for CSR scalar and ELL

Initialize $r_0, p_0, x_0, \sigma_0, \tau_0; j := 0$

while $(\tau_j > \tau_{\max})$

- O1. SPMV
- O2. DOT
- O3. AXPY
- O4. AXPY
- O5. DOTproduct
- O6. Scalar op
- O7. Scalar op
- O8. XPAY (AXPY-like)
- O9. Vector 2-norm (sqrt)
- $j := j + 1$

endwhile



Fusion of CUDA kernels for CSR vector

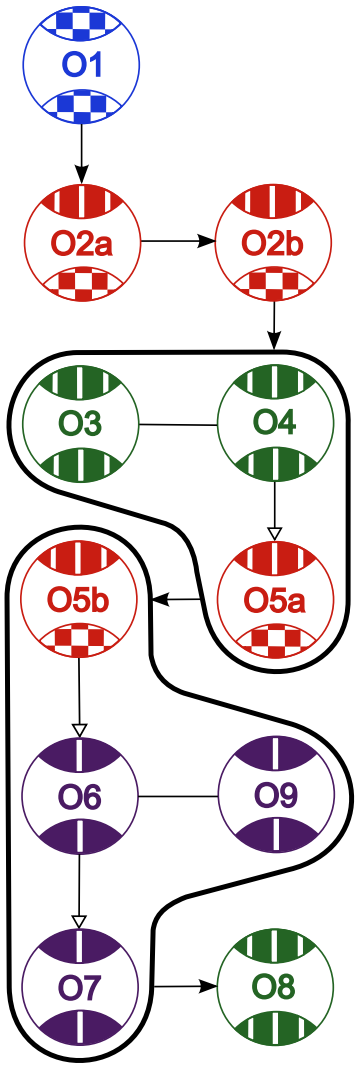
Initialize $r_0, p_0, x_0, \sigma_0, \tau_0; j := 0$

while $(\tau_j > \tau_{\max})$

- O1. SPMV
- O2. DOT
- O3. AXPY
- O4. AXPY
- O5. DOTproduct
- O6. Scalar op
- O7. Scalar op
- O8. XPAY (AXPY-like)
- O9. Vector 2-norm (sqrt)

$j := j + 1$

endwhile



Outline

Dynamic Parallelism

- Development of Dynamic Parallelism
  - Technology recently introduced in the CUDA programming mode
  - Available for NVIDIA devices with compute capability 3.5 or higher
- Main features of Dynamic Parallelism
  - Allows CUDA kernels (parent) to launch new kernels (child)
    - The recursion is enabled
    - Useful to adapt the grid size at execution time
  - The synchronization between parent and child kernels occurs
    - Implicitly, at the end of the parent kernel
    - Explicitly, by using `cudaDeviceSynchronize`
  - Given the current DP limits, it is advisable
    - Reduce the depth of the call tree
    - Use iterative before recursive implementations

Dynamic Parallelism

- Development of Dynamic Parallelism
  - Technology recently introduced in the CUDA programming mode
  - Available for NVIDIA devices with compute capability 3.5 or higher
- Main features of Dynamic Parallelism
  - Allows CUDA kernels (parent) to launch new kernels (child)
    - The recursion is enabled
    - Useful to adapt the grid size at execution time
  - The synchronization between parent and child kernels occurs
    - Implicitly, at the end of the parent kernel
    - Explicitly, by using `cudaDeviceSynchronize`
  - Given the current DP limits, it is advisable
    - Reduce the depth of the call tree
    - Use iterative before recursive implementations

Dynamic Parallelism

- Development of Dynamic Parallelism
  - Technology recently introduced in the CUDA programming mode
  - Available for NVIDIA devices with compute capability 3.5 or higher
- Main features of Dynamic Parallelism
  - Allows CUDA kernels (parent) to launch new kernels (child)
    - The recursion is enabled
    - Useful to adapt the grid size at execution time
  - The synchronization between parent and child kernels occurs
    - Implicitly, at the end of the parent kernel
    - Explicitly, by using `cudaDeviceSynchronize`
  - Given the current DP limits, it is advisable
    - Reduce the depth of the call tree
    - Use iterative before recursive implementations

Dynamic Parallelism

- Development of Dynamic Parallelism
  - Technology recently introduced in the CUDA programming mode
  - Available for NVIDIA devices with compute capability 3.5 or higher
- Main features of Dynamic Parallelism
  - Allows CUDA kernels (parent) to launch new kernels (child)
    - The recursion is enabled
    - Useful to adapt the grid size at execution time
  - The synchronization between parent and child kernels occurs
    - Implicitly, at the end of the parent kernel
    - Explicitly, by using `cudaDeviceSynchronize`
  - Given the current DP limits, it is advisable
    - Reduce the depth of the call tree
    - Use iterative before recursive implementations

cudaSdot_DP

- The definition of cudaSdot includes a routine (loop of kernels), according to the implementation of cudaSreduction
- The cudaSreduction implementation should be modified
  - The loop of kernels should be changed by a kernel with a loop
  - The final value has to written by a kernel with one block
  - There is no sense this kernel computes all the reduction
  - An alternative is to use two kernels:
    - Kernel with several blocks, which executes a reduction loop
    - Kernel with one block to compute the last reduction
  - An additional optimization:
    - Double the size of the consecutive data processed by a block
    - The coalescent access is maintained into the block

cudaSdot_DP

- The definition of cudaSdot includes a routine (loop of kernels), according to the implementation of cudaSreduction
- The cudaSreduction implementation should be modified
  - The loop of kernels should be changed by a kernel with a loop
  - The final value has to written by a kernel with one block
  - There is no sense this kernel computes all the reduction
  - An alternative is to use two kernels:
    - Kernel with several blocks, which executes a reduction loop
    - Kernel with one block to compute the last reduction
  - An additional optimization:
    - Double the size of the consecutive data processed by a block
    - The coalescent access is maintained into the block

cudaSdot_DP

- The definition of cudaSdot includes a routine (loop of kernels), according to the implementation of cudaSreduction
- The cudaSreduction implementation should be modified
  - The loop of kernels should be changed by a kernel with a loop
  - The final value has to written by a kernel with one block
  - There is no sense this kernel computes all the reduction
  - An alternative is to use two kernels:
    - Kernel with several blocks, which executes a reduction loop
    - Kernel with one block to compute the last reduction
  - An additional optimization:
    - Double the size of the consecutive data processed by a block
    - The coalescent access is maintained into the block

cudaSdot_DP

- The definition of cudaSdot includes a routine (loop of kernels), according to the implementation of cudaSreduction
- The cudaSreduction implementation should be modified
  - The loop of kernels should be changed by a kernel with a loop
  - The final value has to written by a kernel with one block
  - There is no sense this kernel computes all the reduction
  - An alternative is to use two kernels:
    - Kernel with several blocks, which executes a reduction loop
      $\Rightarrow$ 256 blocks each with 192 threads
    - Kernel with one block to compute the last reduction
      $\Rightarrow$ 1 block with 256 threads
  - An additional optimization:
    - Double the size of the consecutive data processed by a block
    - The coalescent access is maintained into the block

cudaSdot_DP

```
1  __global__ void cudaSreduction_DP_loop (int n, float *in, float *out) {
2    extern __shared__ float vtmp[];
3    // Each thread loads two elements from each chunk
4    // from global to shared memory
5    unsigned int tid     = threadIdx.x;                    // block thread index
6    unsigned int NumBlk  = gridDim.x;                      // = 256
7    unsigned int BlkSize = blockDim.x;                     // = 192
8    unsigned int Chunk   = NumBlk * BlkSize;               // Size of the grid data
9    unsigned int i       = blockIdx.x * BlkSize + tid;
10
11   // Reduce from n to NumBlk * BlkSize elements. Each thread
12   // operates with two elements of each chunk
13   vtmp[tid] = 0;
14   while (i < n) {
15     vtmp[tid] += in[i];
16
17     i         += Chunk;
18   } __syncthreads();
19
20   // Reduce from BlkSize=192 elements to 96, 48, 24, 12, 6, 3 and 1
21   if (tid < 96) { vtmp[tid] += vtmp[tid + 96]; } __syncthreads();
22   if (tid < 48) { vtmp[tid] += vtmp[tid + 48]; } __syncthreads();
23   if (tid < 24) {
24     volatile float *vtmp2   = vtmp;
25     vtmp2[tid] += vtmp2[tid + 24]; vtmp2[tid] += vtmp2[tid + 12];
26     vtmp2[tid] += vtmp2[tid + 6 ]; vtmp2[tid] += vtmp2[tid + 3 ];
27   }
28
29   // Write result for this block to global mem
30   if (tid == 0) out[blockIdx.x] = vtmp[0] + vtmp[1] + vtmp[2];
31 }
```

cudaSdot_DP

```
1  __global__ void cudaSreduction_DP_final (float *in_out) {
2    extern __shared__ float vtmp[];
3
4    // Each thread loads one element from global to shared mem
5    unsigned int   tid = threadIdx.x;
6    volatile float *vtmp2 = vtmp;
7
8    vtmp[tid] = in_out[tid]; __syncthreads();
9
10   // Reduce from 256 elements to 128, 64, 32, 16, 8, 2 and 1
11   if (tid < 128) { vtmp[tid] += vtmp[tid + 128]; } __syncthreads();
12   if (tid < 64)  { vtmp[tid] += vtmp[tid + 64 ]; } __syncthreads();
13   if (tid < 32)  {
14     vtmp2[tid] += vtmp2[tid + 32]; vtmp2[tid] += vtmp2[tid + 16];
15     vtmp2[tid] += vtmp2[tid + 8 ]; vtmp2[tid] += vtmp2[tid + 4 ];
16     vtmp2[tid] += vtmp2[tid + 2 ]; vtmp2[tid] += vtmp2[tid + 1 ];
17   }
18
19   // Write result for this block to global mem
20   if (tid == 0) in_out[blockIdx.x] = *vtmp;
21 }
```

cudaSdot_DP

- The definition of cudaSdot includes a routine (loop of kernels), according to the implementation of cudaSreduction
- The cudaSreduction implementation should be modified
  - The loop of kernels should be changed by a kernel with a loop
  - The final value has to written by a kernel with one block
  - There is no sense this kernel computes all the reduction
  - An alternative is to use two kernels:
    - Kernel with several blocks, which executes a reduction loop
    - Kernel with one block to compute the last reduction
  - An additional optimization:
    - Double the size of the consecutive data processed by a block
    - The coalescent access is maintained into the block

cudaSdot_DP

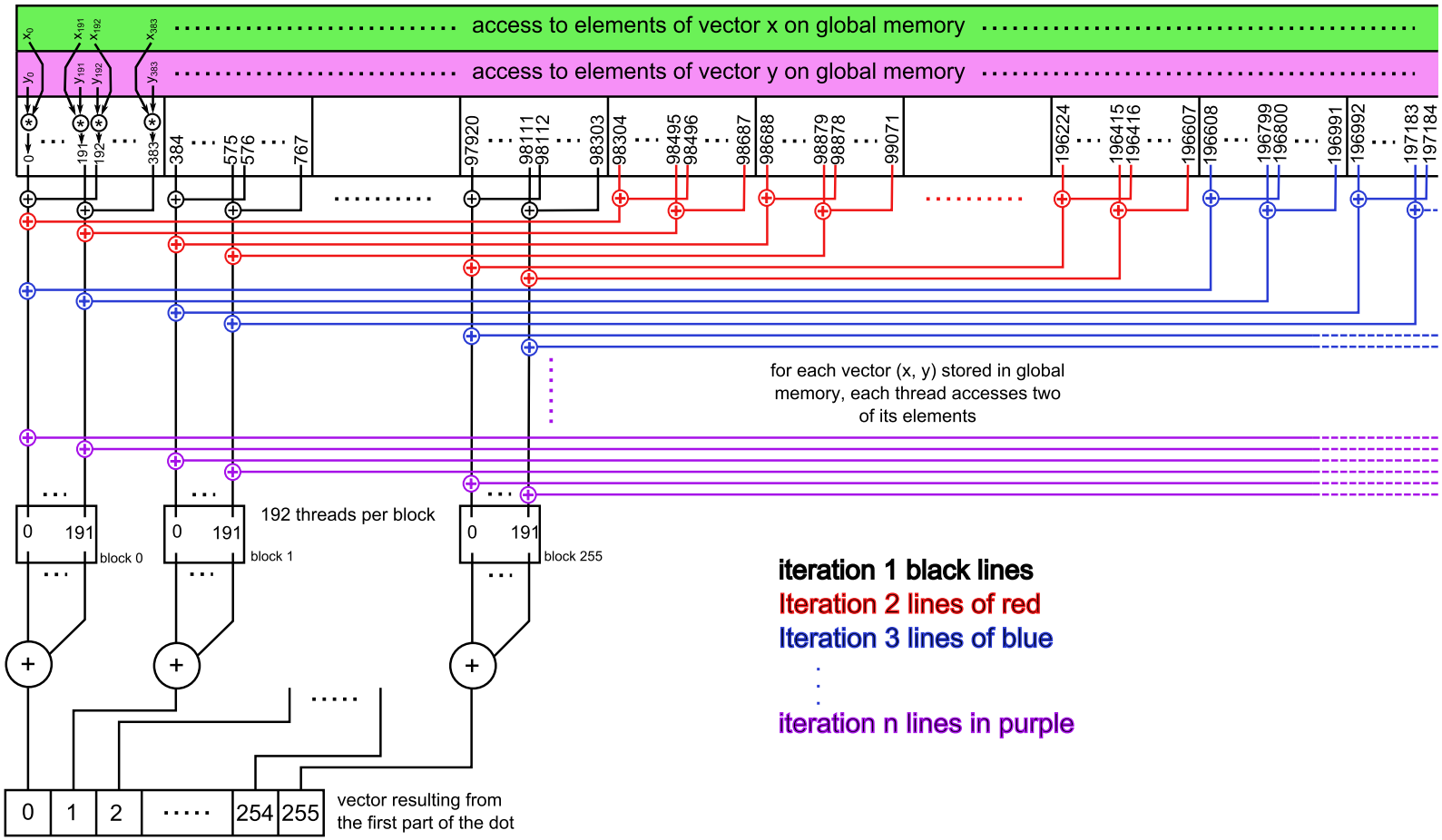
```
1  __global__ void cudaSreduction_DP_loop_opt (int n, float *in, float *out) {
2    extern __shared__ float vtmp[];
3    // Each thread loads two elements from each chunk
4    // from global to shared memory
5    unsigned int tid     = threadIdx.x;                    // block thread index
6    unsigned int NumBlk  = gridDim.x;                      // = 256
7    unsigned int BlkSize = blockDim.x;                     // = 192
8    unsigned int Chunk   = NumBlk * (2 * BlkSize);         // Size of the grid data
9    unsigned int i       = blockIdx.x * (2 * BlkSize) + tid;
10
11   // Reduce from n to NumBlk * BlkSize elements. Each thread
12   // operates with two elements of each chunk
13   vtmp[tid] = 0;
14   while (i < n) {
15     vtmp[tid] += in[i];
16     vtmp[tid] += (i+BlkSize < n) ? (in[i+BlkSize]): 0;
17     i         += Chunk;
18   } __syncthreads();
19
20   // Reduce from BlkSize=192 elements to 96, 48, 24, 12, 6, 3 and 1
21   if (tid < 96) { vtmp[tid] += vtmp[tid + 96]; } __syncthreads();
22   if (tid < 48) { vtmp[tid] += vtmp[tid + 48]; } __syncthreads();
23   if (tid < 24) {
24     volatile float *vtmp2   = vtmp;
25     vtmp2[tid] += vtmp2[tid + 24]; vtmp2[tid] += vtmp2[tid + 12];
26     vtmp2[tid] += vtmp2[tid + 6 ]; vtmp2[tid] += vtmp2[tid + 3 ];
27   }
28
29   // Write result for this block to global mem
30   if (tid == 0) out[blockIdx.x] = vtmp[0] + vtmp[1] + vtmp[2];
31 }
```

cudaSdot_DP

```
1  __global__ void cudaSdot_DP_loop_opt (int n, float *x, , float *y, float *out) {
2    extern __shared__ float vtmp[];
3    // Each thread loads two elements from each chunk
4    // from global to shared memory
5    unsigned int tid     = threadIdx.x;                    // block thread index
6    unsigned int NumBlk  = gridDim.x;                      // = 256
7    unsigned int BlkSize = blockDim.x;                     // = 192
8    unsigned int Chunk   = NumBlk * (2 * BlkSize);         // Size of the grid data
9    unsigned int i       = blockIdx.x * (2 * BlkSize) + tid;
10
11   // Reduce from n to NumBlk * BlkSize elements. Each thread
12   // operates with two elements of each chunk
13   vtmp[tid] = 0;
14   while (i < n) {
15     vtmp[tid] += x[i]*y[i];
16     vtmp[tid] += (i+BlkSize < n) ? (x[i+BlkSize]*y[i+BlkSize]): 0;
17     i         += Chunk;
18   } __syncthreads();
19
20   // Reduce from BlkSize=192 elements to 96, 48, 24, 12, 6, 3 and 1
21   if (tid < 96) { vtmp[tid] += vtmp[tid + 96]; } __syncthreads();
22   if (tid < 48) { vtmp[tid] += vtmp[tid + 48]; } __syncthreads();
23   if (tid < 24) {
24     volatile float *vtmp2   = vtmp;
25     vtmp2[tid] += vtmp2[tid + 24]; vtmp2[tid] += vtmp2[tid + 12];
26     vtmp2[tid] += vtmp2[tid + 6 ]; vtmp2[tid] += vtmp2[tid + 3 ];
27   }
28
29   // Write result for this block to global mem
30   if (tid == 0) out[blockIdx.x] = vtmp[0] + vtmp[1] + vtmp[2];
31 }
```

cudaSdot_DP

cudaSdot_DP_loop



cudaSaxpy_DP

- The changes in cudaSdot can also be applied on cudaSaxpy
  - To use a loop to reduce the number of blocks into the grid
  - Double the size of the consecutive data processed by a block
- The optimal results are obtained when a thread processes two elements

```
1  __global__ void cudaSaxpy_DP (int n, float *alpha, float *x, float *y) {
2    int NumBlk  = gridDim.x;                                  // = ceil (n / 256)
3    int BlkSize = blockDim.x;                                 // = 128
4    int i       = blockIdx.x * (2 * BlkSize) + threadIdx.x;
5    int Chunk   = 2 * NumBlk * BlkSize;
6
7    while (i < n) {
8      y[i] +=  *alfa * x[i];
9      if (i + BlkSize < n) y[i + BlkSize] += *alfa * x[i + BlkSize];
10     i += Chunk;
11   }
12 }
```

cudaSaxpy_DP

- The changes in cudaSdot can also be applied on cudaSaxpy
  - To use a loop to reduce the number of blocks into the grid
  - Double the size of the consecutive data processed by a block
- The optimal results are obtained when a thread processes two elements

```
1  __global__ void cudaSaxpy_DP (int n, float *alpha, float *x, float *y) {
2    int NumBlk  = gridDim.x;                                  // = ceil (n / 256)
3    int BlkSize = blockDim.x;                                 // = 128
4    int i       = blockIdx.x * (2 * BlkSize) + threadIdx.x;
5    int Chunk   = 2 * NumBlk * BlkSize;
6
7    while (i < n) {
8      y[i] +=  *alfa * x[i];
9      if (i + BlkSize < n) y[i + BlkSize] += *alfa * x[i + BlkSize];
10     i += Chunk;
11   }
12 }
```

CG algorithm for Dynamic Parallelism

Initialize $r_0, p_0, x_0, \sigma_0, \tau_0; j := 0$

while $(\tau_j > \tau_{\max})$

- O1. SPMV
- O2. DOT
- O3. AXPY
- O4. AXPY
- O5. DOTproduct
- O6. Scalar op
- O7. Scalar op
- O8. XPAY (AXPY-like)
- O9. Vector 2-norm (sqrt)
- $j := j + 1$
- `cudaDeviceSynchronize`

endwhile

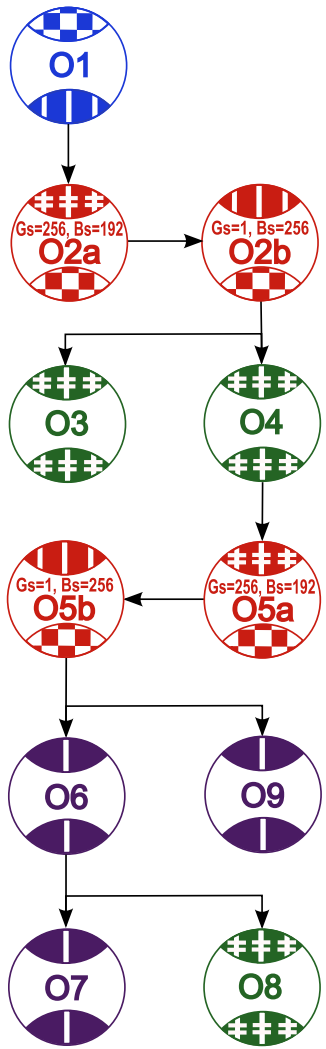
Dependendy Graph for DP CUDA kernels of CG

Initialize $r_0, p_0, x_0, \sigma_0, \tau_0; j := 0$

while $(\tau_j > \tau_{\max})$

- O1. SPMV
- O2. DOT
- O3. AXPY
- O4. AXPY
- O5. DOTproduct
- O6. Scalar op
- O7. Scalar op
- O8. XPAY (AXPY-like)
- O9. Vector 2-norm (sqrt)
- $j := j + 1$
- `cudaDeviceSynchronize`

endwhile



Fusion of DP CUDA kernels for CSR scalar and ELL

Initialize $r_0, p_0, x_0, \sigma_0, \tau_0; j := 0$

while $(\tau_j > \tau_{\max})$

O1. SPMV

O2. DOT

O3. AXPY

O4. AXPY

O5. DOTproduct

O6. Scalar op

O7. Scalar op

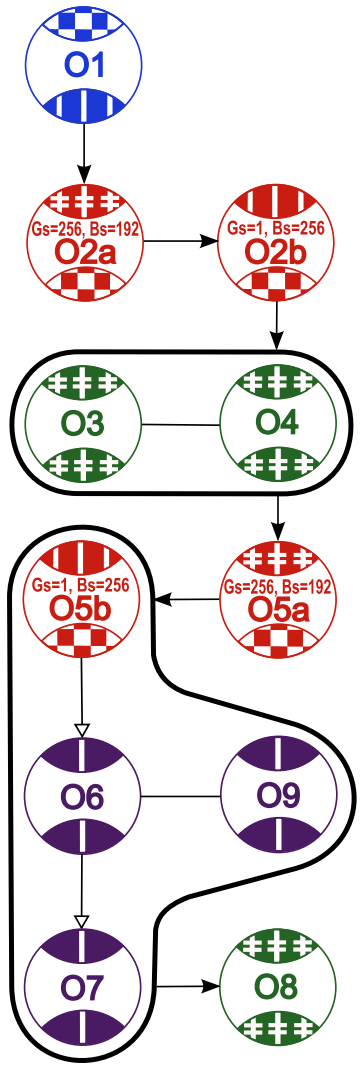
O8. XPAY (AXPY-like)

O9. Vector 2-norm (sqrt)

$j := j + 1$

`cudaDeviceSynchronize`

endwhile



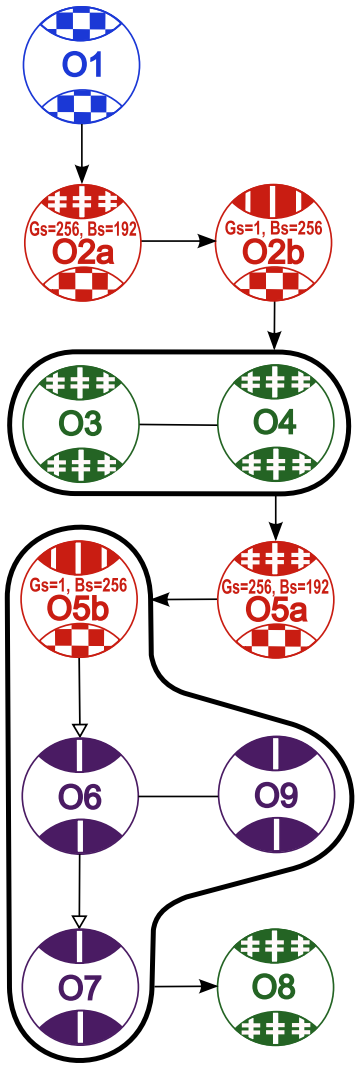
Fusion of DP CUDA kernels for CSR vector

Initialize $r_0, p_0, x_0, \sigma_0, \tau_0; j := 0$

while $(\tau_j > \tau_{\max})$

- O1. SPMV
- O2. DOT
- O3. AXPY
- O4. AXPY
- O5. DOTproduct
- O6. Scalar op
- O7. Scalar op
- O8. XPAY (AXPY-like)
- O9. Vector 2-norm (sqrt)
- $j := j + 1$
- `cudaDeviceSynchronize`

endwhile



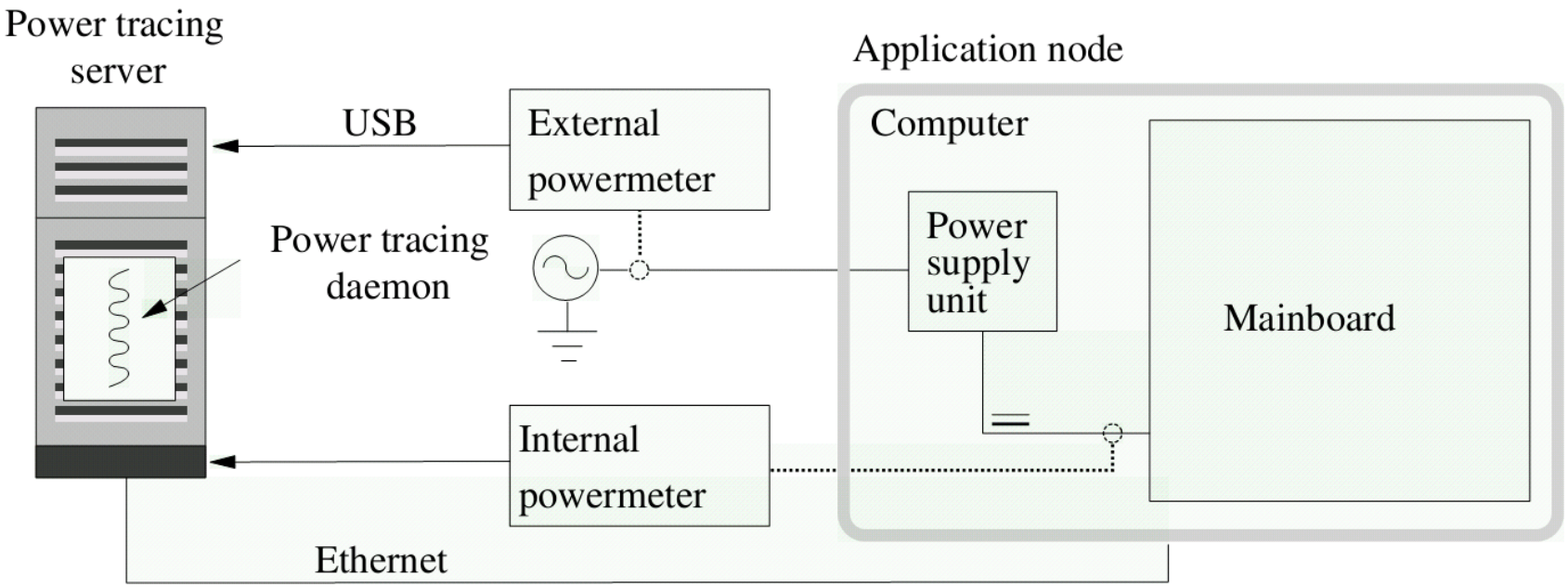
Outline

Hardware Platform

- Target Platform
  - CentOS release 6.2 with kernel 2.6.32 with CUDA v5.5.0
  - Intel Core i7-3770K CPU (3.5 GHz, four cores) and 16 Gbytes of DDR3 RAM
  - NVIDIA “Kepler” K20c GPU (C.C. 3.5, 706 MHz, 2,496 CUDA cores) with 5 GB of DDR5 RAM
  - CPU-GPU connection via a PCI-e 2.0 bus
- How to trace the power consumption?
  - Using a *National Instruments* data acquisition system
    - NI9205 modules and NIcDAQ-9178 chassis
    - Lines that connect the PSU with motherboard and GPU.
    - The sampling frequency was 1MHz
  - The samples was processed by the power tracing server

Hardware Platform

Power Tracing Environment



Hardware Platform

- Target Platform
  - CentOS release 6.2 with kernel 2.6.32 with CUDA v5.5.0
  - Intel Core i7-3770K CPU (3.5 GHz, four cores) and 16 Gbytes of DDR3 RAM
  - NVIDIA “Kepler” K20c GPU (C.C. 3.5, 706 MHz, 2,496 CUDA cores) with 5 GB of DDR5 RAM
  - CPU-GPU connection via a PCI-e 2.0 bus
- How to trace the power consumption?
  - Using a *National Instruments* data acquisition system
    - NI9205 modules and NIcDAQ-9178 chassis
    - Lines that connect the PSU with motherboard and GPU.
    - The sampling frequency was 1MHz
  - The samples was processed by the power tracing server

Arithmetics

- CG Implementations
  - $x_0 = 0, b = A * ones(n, 1), maxiter = 1000$
  - IEEE single precision arithmetic ($\epsilon = 10^{-5}$)
    - On GPUs, mixed SP-DP with iterative refinement improves execution time and energy consumption
    - SP is the computational key of mixed SP-DP
- Benchmark Matrices

	Matrix	Acronym	n_z	n	n_z/n
UFMC	BMWCRA1_1	bmw	10,641,602	148,770	71.53
	CRANKSEG_2	crank	14,148,858	63,838	221.63
	F1	F1	26,837,113	343,791	78.06
	INLINE_1	inline	38,816,170	503,712	77.06
	LDOOR	ldoor	42,493,817	952,203	44.62
	AUDIKW_1	audi	77,651,847	943,645	82.28
	A252	A252	111,640,032	16,003,001	6.94

Arithmetics

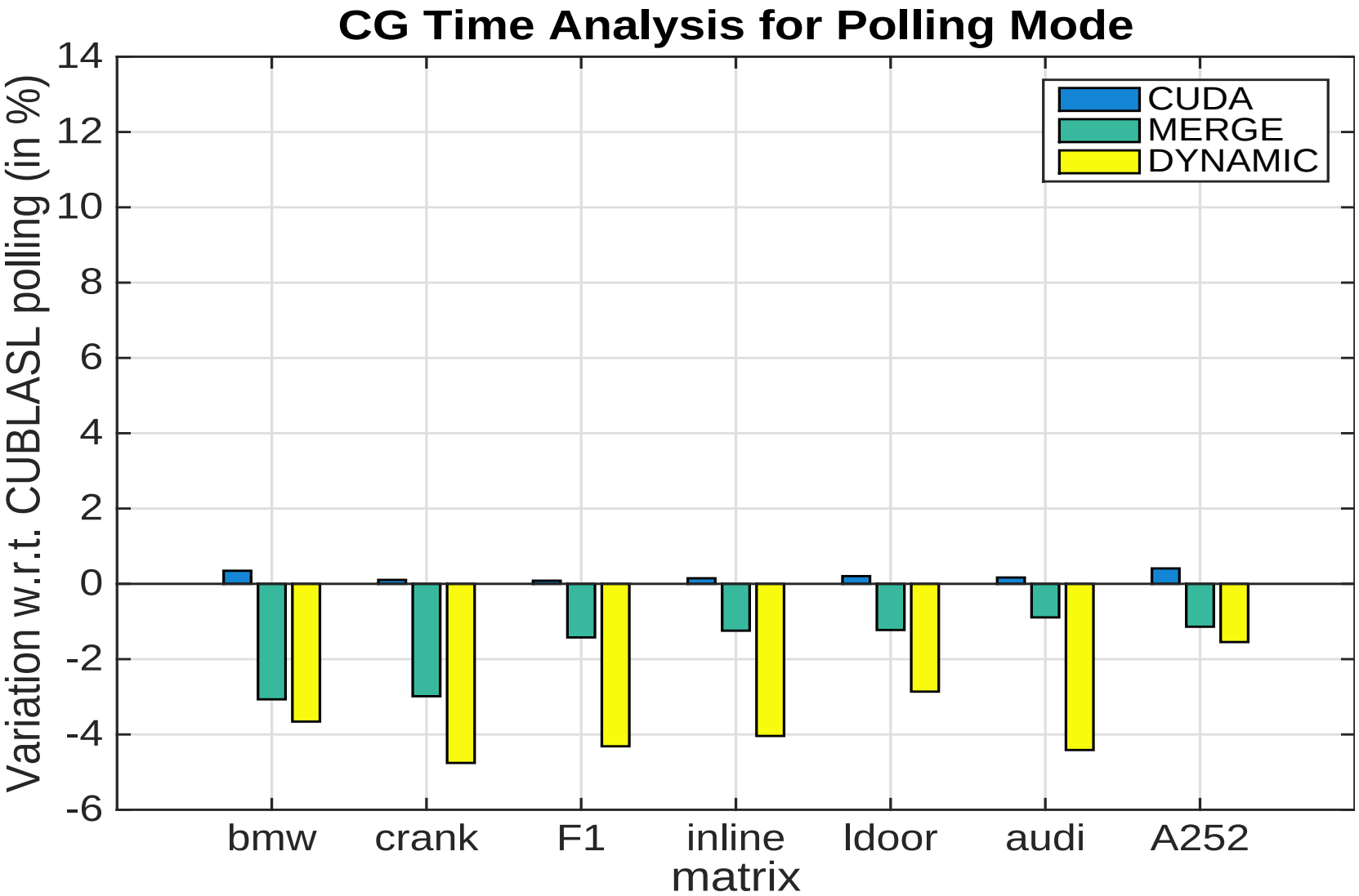
- CG Implementations
  - $x_0 = 0, b = A * ones(n, 1), maxiter = 1000$
  - IEEE single precision arithmetic ($\epsilon = 10^{-5}$)
    - On GPUs, mixed SP-DP with iterative refinement improves execution time and energy consumption
    - SP is the computational key of mixed SP-DP
- Benchmark Matrices

	Matrix	Acronym	n_z	n	n_z/n
UFMC	BMWCRA1_1	bmw	10,641,602	148,770	71.53
	CRANKSEG_2	crank	14,148,858	63,838	221.63
	F1	F1	26,837,113	343,791	78.06
	INLINE_1	inline	38,816,170	503,712	77.06
	LDOOR	ldoor	42,493,817	952,203	44.62
	AUDIKW_1	audi	77,651,847	943,645	82.28
	A252	A252	111,640,032	16,003,001	6.94

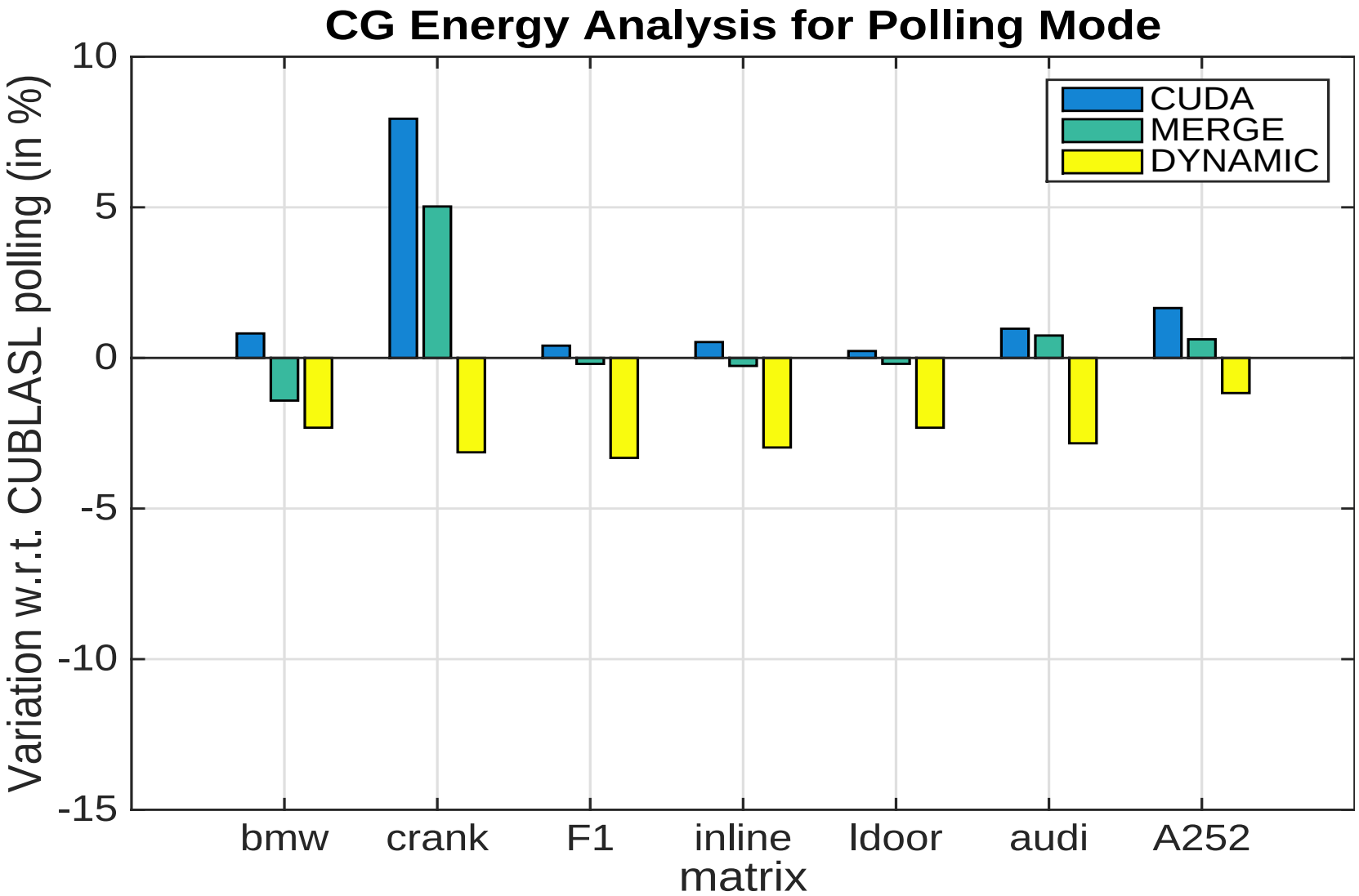
Implementations of the CG solver

- For SPMV, we have tested the three described CUDA kernels
- CUBLASL, uses the legacy programming interface of CUBLAS
  - The scalars are stored in CPU (some transfers are required)
  - The vector operations are performed by using CUBLAS kernels
- CUDA replaces CUBLAS kernels by CUDA kernels
  - The scalars are stored in GPU
  - The vector operations are performed by CUDA kernels
- MERGE applies the fusions defined on the dependency graph
  - New CUDA kernels implement the kernel fusion
- DYNAMIC applied the fusions defined on the dependency graph
  - The CG CUDA kernels are launched from a CUDA kernel

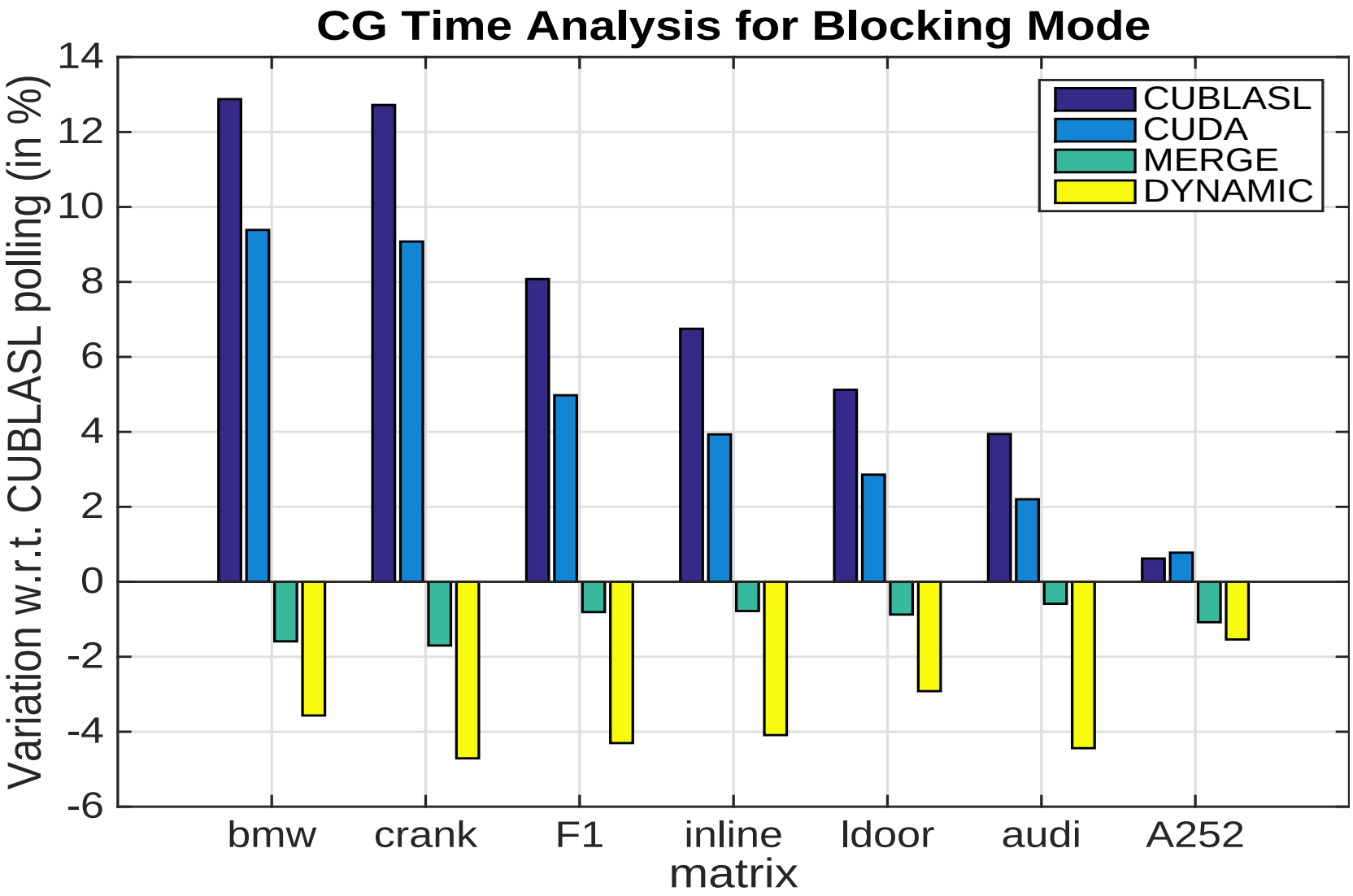
Results for the CSR-vector kernel



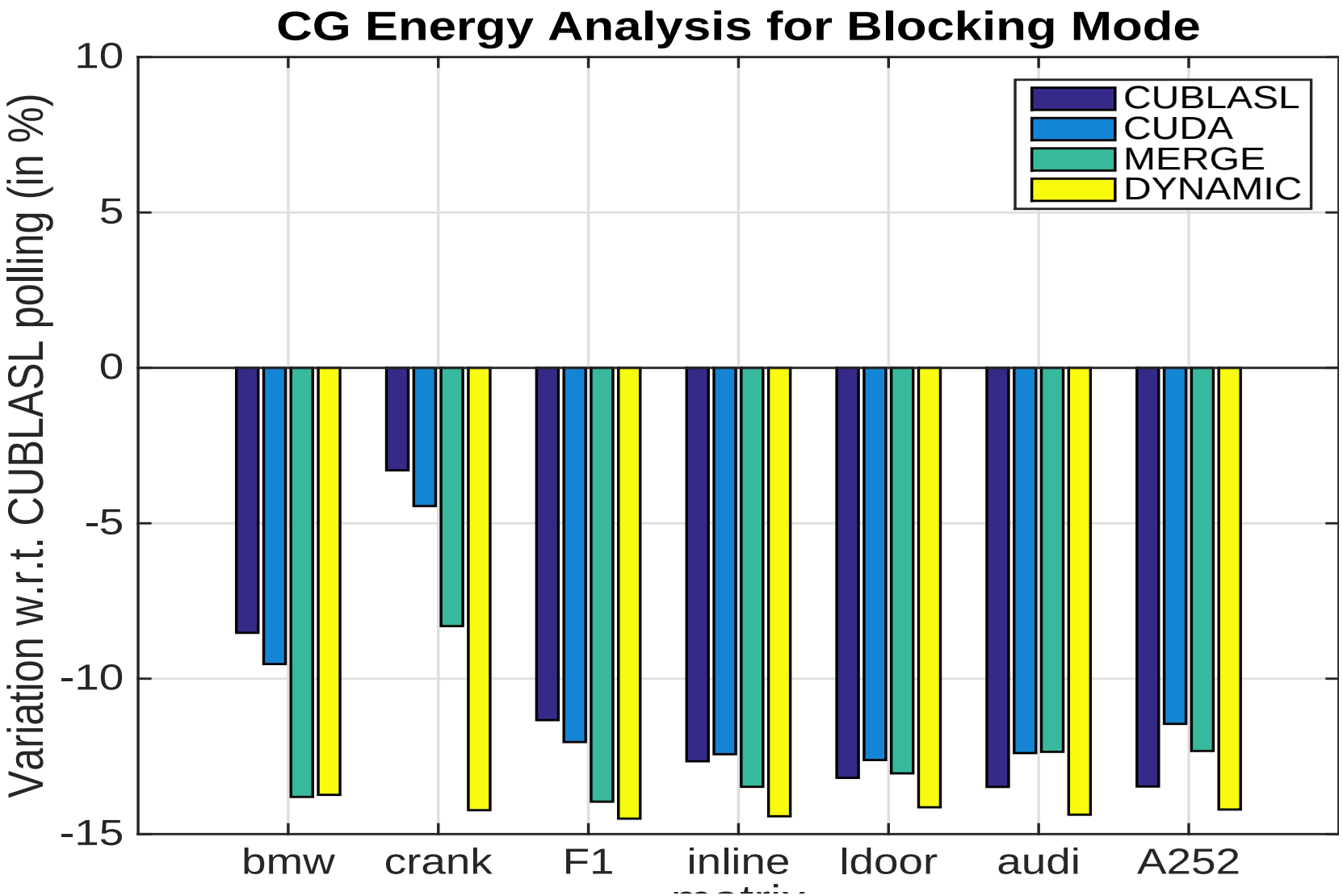
Results for the CSR-vector kernel



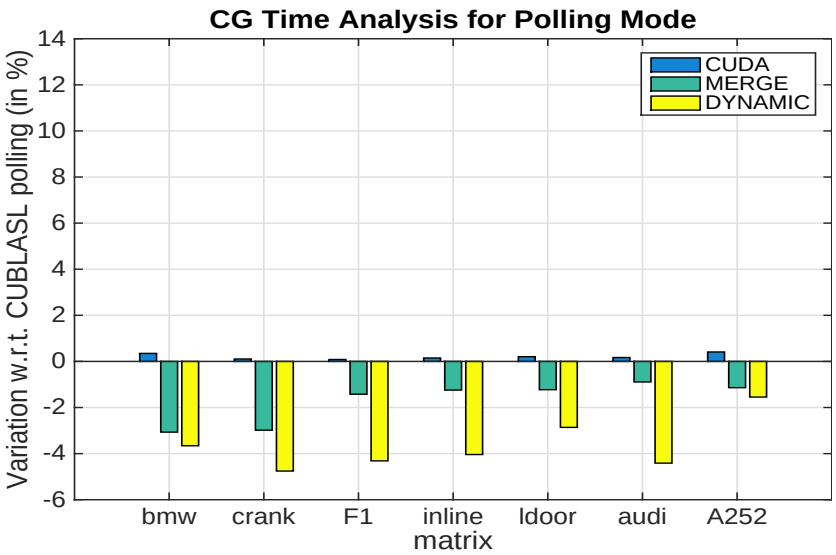
Results for the CSR-vector kernel

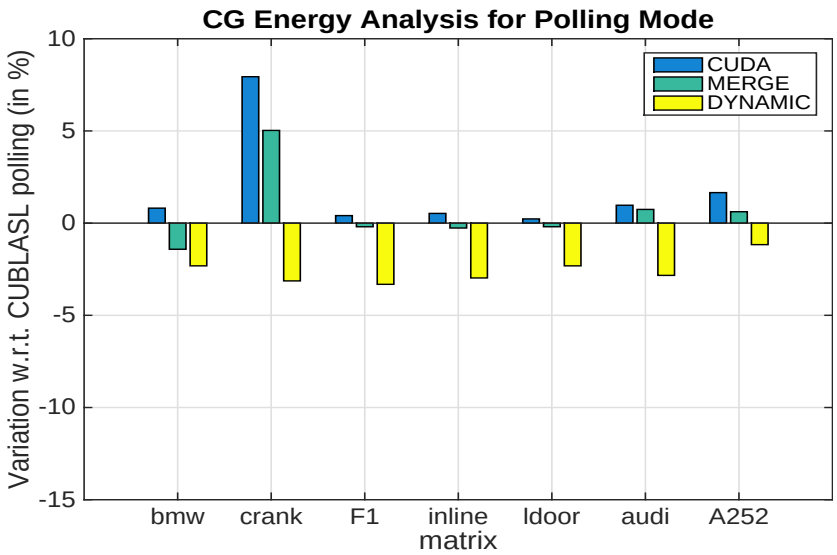


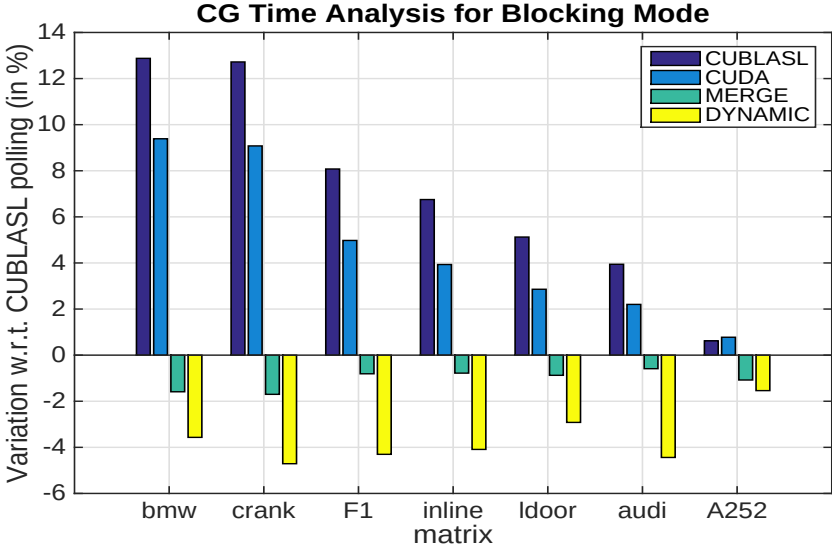
Results for the CSR-vector kernel

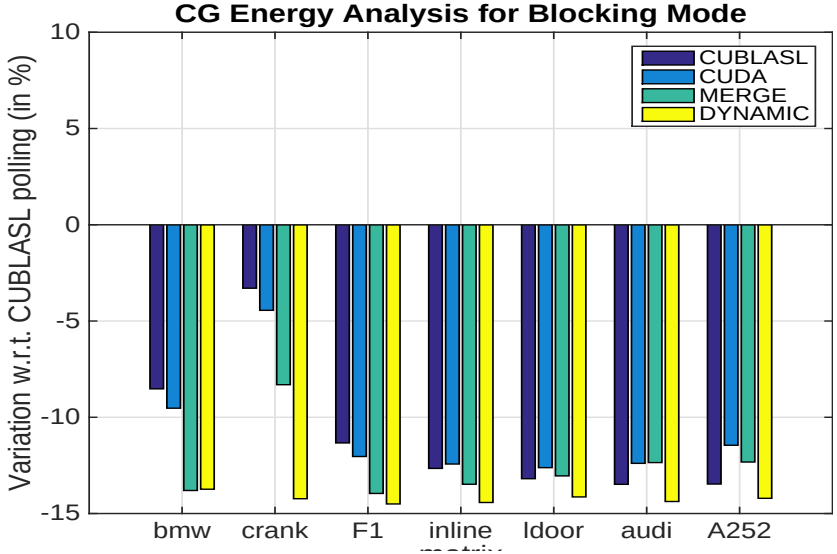


Results for the CSR-vector kernel









Results for the CSR-vector kernel

Summary of the variations (in %) w.r.t. CUBLASL-polling

CUDA mode	CG solver Implement.	Time Min	Time Max	Time Avg.	Energy Min	Energy Max	Energy Avg.
Polling	CUBLASL	0.00	0.00	0.00	0.00	0.00	0.00
	CUDA	0.08	0.41	0.21	0.23	7.94	1.79
	MERGE	-3.07	-0.89	-1.71	-1.42	5.03	0.62
	DYNAMIC	-4.76	-1.54	-3.65	-3.32	-1.17	-2.58
Blocking	CUBLASL	0.62	12.88	7.16	-3.30	-13.48	-10.85
	CUDA	0.78	9.39	4.74	-12.62	-4.45	-10.70
	MERGE	-1.70	-0.59	-1.06	-13.96	-8.31	-12.47
	DYNAMIC	-4.71	-1.54	-3.65	-14.50	-13.74	-14.23

Conclusions and Future Work

- We have exploited DP to implement a CUDA-CG solver
  - The CPU only launches a simple CUDA kernel
  - The GPU is in charge of executing the complete solver
  - The CPU can execute other tasks or be simply put to sleep
- We have redesigned two CUDA kernels
  - cudaSdot_DP reduces the depth of the call tree
  - cudaSaxpy_DP improves the performances of the cudaSaxpy
- Evaluation of the DP version of the CUDA-CG solver:
  - Reduce the execution time by 3.65% in both CUDA modes
  - Reduce the energy consumption by 14.23% in blocking mode
- **Future Work:** Extent the DP to more complex solvers (PCG, BiCG, . . .)

Conclusions and Future Work

- We have exploited DP to implement a CUDA-CG solver
  - The CPU only launches a simple CUDA kernel
  - The GPU is in charge of executing the complete solver
  - The CPU can execute other tasks or be simply put to sleep
- We have redesigned two CUDA kernels
  - cudaSdot_DP reduces the depth of the call tree
  - cudaSaxpy_DP improves the performances of the cudaSaxpy
- Evaluation of the DP version of the CUDA-CG solver:
  - Reduce the execution time by 3.65% in both CUDA modes
  - Reduce the energy consumption by 14.23% in blocking mode
- **Future Work:** Extent the DP to more complex solvers (PCG, BiCG, . . .)

Conclusions and Future Work

- We have exploited DP to implement a CUDA-CG solver
  - The CPU only launches a simple CUDA kernel
  - The GPU is in charge of executing the complete solver
  - The CPU can execute other tasks or be simply put to sleep
- We have redesigned two CUDA kernels
  - cudaSdot_DP reduces the depth of the call tree
  - cudaSaxpy_DP improves the performances of the cudaSaxpy
- Evaluation of the DP version of the CUDA-CG solver:
  - Reduce the execution time by 3.65% in both CUDA modes
  - Reduce the energy consumption by 14.23% in blocking mode
- **Future Work:** Extent the DP to more complex solvers (PCG, BiCG, . . .)

Conclusions and Future Work

- We have exploited DP to implement a CUDA-CG solver
  - The CPU only launches a simple CUDA kernel
  - The GPU is in charge of executing the complete solver
  - The CPU can execute other tasks or be simply put to sleep
- We have redesigned two CUDA kernels
  - cudaSdot_DP reduces the depth of the call tree
  - cudaSaxpy_DP improves the performances of the cudaSaxpy
- Evaluation of the DP version of the CUDA-CG solver:
  - Reduce the execution time by 3.65% in both CUDA modes
  - Reduce the energy consumption by 14.23% in blocking mode
- **Future Work:** Extent the DP to more complex solvers (PCG, BiCG, . . .)

Thanks for your attention !

Questions ?