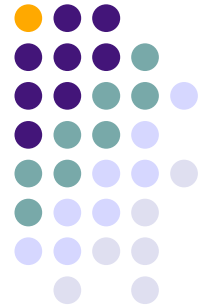
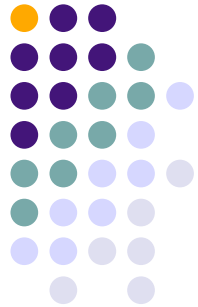


Reformulated Conjugate Gradient for the Energy-Aware Solution of Linear Systems on GPUs

José I. Aliaga

Reformulated Conjugate Gradient for the Energy-Aware Solution of Linear Systems on GPUs



- Universidad Jaime I (Castellón, Spain)
  - José I. Aliaga
  - Joaquín Pérez
  - Enrique S. Quintana-Ortí

UNIVERSITAT JAUME·I

- Innovative and Computing Lab (Univ. Tennessee, USA)
  - Hartwig Antz

ICL UT

THE UNIVERSITY of TENNESSEE

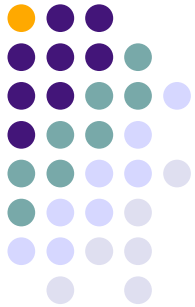
Concurrency and energy efficiency

2010 PFLOPS (10^{15} flops/sec.)

2010 JUGENE

- 10^9 core level
  (PowerPC 450, 850MHz → 3.4 GFLOPS)
- 10^1 node level
  (Quad-Core)
- 10^5 cluster level
  (73.728 nodes)

Concurrency and energy efficiency



2010 PFLOPS (10^{15} flops/sec.)

2010 JUGENE

- 10^9 core level
  (PowerPC 450, 850MHz → 3.4 GFLOPS)
- 10^1 node level
  (Quad-Core)
- 10^5 cluster level
  (73.728 nodes)

2020 EFLOPS (10^{18} flops/sec.)

- $10^{9.5}$ core level
- 10^3 node level!
- $10^{5.5}$ cluster level

Concurrency and energy efficiency

- Green500/Top500 (November 2010 – June 2013)

Nov-2010

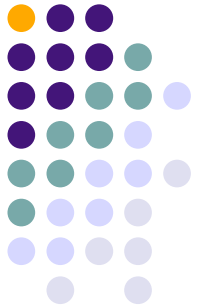
Rank Green/Top	Site, Computer	#Cores	MFLOPS/W	LINPACK (TFLOPS)	MW to EXAFLOPS?
1/115	NNSA/SC Blue Gene/Q Prototype	8.192	1.684'20	65'35	593'75
11/1	NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C	186.368	635'15	2.566'00	1.574'43

16 3 13

Jun-2013

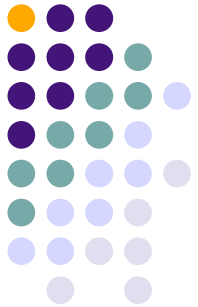
Rank Green/Top	Site, Computer	#Cores	MFLOPS/W	LINPACK (TFLOPS)	MW to EXAFLOPS?
1/467	Eurotech Aurora HPC 10-20, Xeon E5-2687W 8C 3.100GHz, Infiniband QDR, NVIDIA K20	2,688	3,208'83	98'51	311'64
20/1	Tianhe-2, TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.2GHz, TH Express-2, Intel Xeon Phi 31S1P	3,120,000	1,901'54	33,862'70	525'89

Concurrency and energy efficiency



- Reduce energy consumption is mandatory!
- High performance computing (HPC) initiatives
  - Hardware features energy saving mechanisms
  - Coprocessors improve the performances
- How to leverage the power-saving mechanisms without sacrificing application's performance
  - Scientific apps are in general energy oblivious
  - Reformulation of the numerical methods

Objective

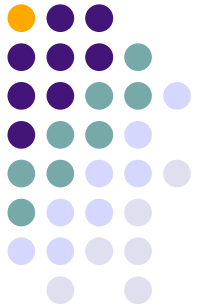


- Energy-aware implementation on GPU of the Conjugate Gradient (CG) method

- Efficient use of the power saving mechanisms
  - Reducing the energy consumption
  - Maintaining the runtime performance

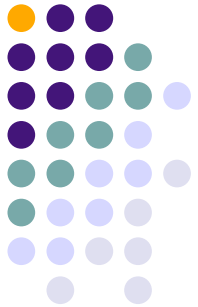
- Redesign the traditional CG method
  - Merging nVidia´s CUDA kernels

Outline



- Introduction
- Solving Sparse SPD Linear Systems (CG)
- Hardware Setup & Benchmark Matrices
- CG-CUBLAS & CG-CUDA Implementations
  - Sparse matrix-vector product
  - BLAS1 operations (dot product)
- Merging CUDA kernels on CG
- Experimental Results
- Conclusions & Future Work

Conjugate Gradient Method



$x_0 := 0$ // or any other initial guess

$r_0 := b - Ax_0$, $\quad d_0 := r_0$

$\beta_0 := r_0^T r_0$, $\quad \tau_0 := \| r_0 \|_2 = \sqrt{\beta_0}$

$k := 0$

while $(k < maxiter)$ & $(\tau_k > maxres)$

- $z_k := Ad_k$
- $\rho_k := \beta_k / d_k^T z_k$
- $x_{k+1} := x_k + \rho_k d_k$
- $r_{k+1} := r_k - \rho_k z_k$
- $\beta_{k+1} := r_{k+1}^T r_{k+1}$
- $\alpha_k := \beta_{k+1} / \beta_k$
- $d_{k+1} := r_{k+1} + \alpha_k d_k$
- $\tau_{k+1} := \| r_{k+1} \|_2 = \sqrt{\beta_{k+1}}$
- $k := k + 1$

end

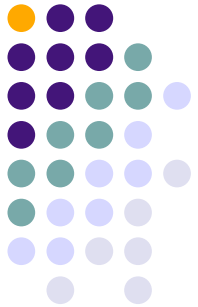
sparse matrix-vector products

dot products

saxpy-like operations

scalar operations

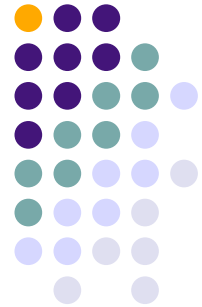
Arithmetic and Hardware Platform



- CG implementations
  - SP is the computational key of mixed SP-DP
  - $\varepsilon = 10^{-5}$, *maxiter* = 1000

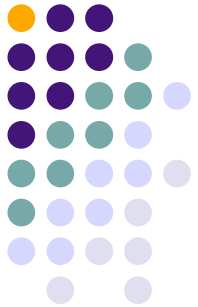
- Hardware platform
  - Intel Core i7-3770K (4 cores at 3.5 GHz) with 16 GB of RAM
  - NVIDIA GeForce GTX480 GPU (1.4 GHz, 1.5 GB)
  - CUDA 4.0 and the legacy CUBLAS API
  - blockSize = 256

Benchmark matrices



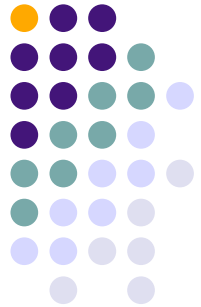
Source	Acronym	Matrix	#nonzeros (nz)	Size (n)	nz/n
UFMC	AUDI	audikw_1	77651847	943695	82,28
	BMW	bmwcra_1	10644002	148770	71,55
	CRANK	crankseg_2	14148858	63838	221,64
	F1	F1	26837113	343791	78,06
	INLINE	inline_1	36816342	503712	73,09
	LDOOR	ldoor	46522475	952203	48,86
Laplace	A100	A100	6940000	1000000	6,94
	A126	A126	13907370	2000376	6,95
	A159	A159	27986067	4019679	6,96
	A200	A200	55760000	8000000	6,97
	A252	A252	111640032	16003008	6,98

CG-CUBLAS Implementation



```
k = 0;
while( ( k < maxiter ) && ( res > epsilon ) ) {
  SSpMV <<<Gs,Bs>>> ( n, rowA, colA, valA, d, z );
                                          // G z := A * d
  tmp = cublasSdot ( n, d, 1, z, 1 );     // G tmp := d' * z
  rho = beta / tmp;                       // C rho := beta / tmp
  gamma = beta;                           // C gamma := beta
  cublasSaxpy (n,  rho, d, 1, x, 1 );     // G x := x + rho * d
  cublasSaxpy (n, -rho, z, 1, r, 1 );     // G r := r - rho * z
  beta = cublasSdot( n, r, 1, r, 1 );     // G beta := r' * r
  alpha = beta / gamma;                   // C alpha := beta/gamma
  cublasSscal (n, alpha, d, 1 );          // G d := alpha * d
  cublasSaxpy (n, one, r, 1, d, 1 );      // G d := d + r
  res = sqrt( beta );                     // C res := sqrt(beta)
  k++;
}
```

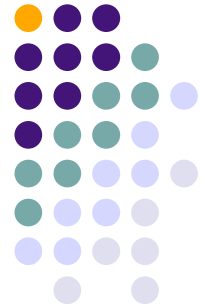
Sparse Matrix-Vector Product (CSR format)



- In the Compressed Sparse Row (CSR) format, a sparse matrix is stored by using 3 vectors.
- A sparse dot product is computed for each element of the solution of the matrix vector product.

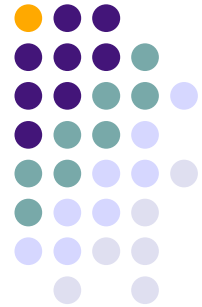
$$\begin{bmatrix} 5 & 2 & 4 & 0 & 0 & 2 & 0 & 5 \\ 3 & 7 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 2 & 0 \end{bmatrix} \xrightarrow{\text{CSR}} \begin{cases} \text{rowptr} = \begin{bmatrix} 0 & 5 & 8 & 10 & 10 & 10 & 11 & 12 & 14 \end{bmatrix} \\ \text{colind} = \begin{bmatrix} 0 & 1 & 2 & 5 & 7 & 0 & 1 & 2 & 2 & 7 & 1 & 1 & 4 & 6 \end{bmatrix} \\ \text{values} = \begin{bmatrix} 5 & 2 & 4 & 2 & 5 & 3 & 7 & 2 & 7 & 5 & 8 & 3 & 1 & 2 \end{bmatrix} \end{cases}$$

Sparse Matrix-Vector Product (CSR format)



- We use nVidia kernels (N. Bell & M. Garland).
- How the sparse dot products are computed:
  - scalar-kernel, each thread computes a dot product.
  - vector-kernel, each warp (32 thr.) computes a dot product, such that a reduction process is required.
- The maximum size of 1D kernels is different
  - scalar-kernel: 65536*blockSize = 16776960 (256)
  - vector-kernel: 65536*blockSize/warpSize = 524280 (256)
- We have defined a 2D variant of the vector-kernel in order to operate on large-scale sparse matrices.

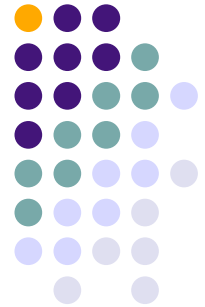
Sparse Matrix-Vector Product (CSR format)



- scalar-kernel

```
int row = blockDim.x * blockIdx.x + threadIdx.x;
// Computation of the sparse dot product
if( row < numrows ){
  float aux = 0.0;
  int frstPos = rowptr[ row ] , lastPos = rowptr[ row + 1 ] ;
  for( int jj = frstPos; jj < lastPos; jj++ ){
    aux += values[ jj ] * x[ colind[ jj ] ];
  }
  y[ row ] = aux;
}
```

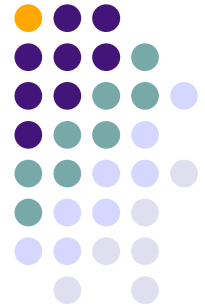
Sparse Matrix-Vector Product (CSR format)



- vector-kernel

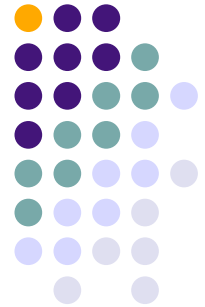
```
int thread_id = blockDim.x * blockIdx.x + threadIdx.x;
int warpSize = 32 , row = thread_id / warpSize;
// Computation of the sparse dot product
if( row < numrows ){
  int frstPos = rowptr[ row ] , lastPos = rowptr[ row + 1 ] ;
  // Computation of the addition in each thread
  vals[ threadIdx.x ] = 0.0;
  for( int jj = frstPos + lane ; jj < lastPos; jj += warpSize ){
    vals[ threadIdx.x ] += values[ jj ] * x[ colind[ jj ] ];
  }
  // Parallel reduction in shared memory
  if ( lane < 16 ) vals[threadIdx.x] += vals[threadIdx.x + 16];
  if ( lane <  8 ) vals[threadIdx.x] += vals[threadIdx.x + 8 ];
  if ( lane <  4 ) vals[threadIdx.x] += vals[threadIdx.x + 4 ];
  if ( lane <  2 ) vals[threadIdx.x] += vals[threadIdx.x + 2 ];
  if ( lane <  1 ) vals[threadIdx.x] += vals[threadIdx.x + 1 ];
  // the first thread writes the result
  if( lane == 0 ) y[ row ] = vals[ threadIdx.x ];
}
```

Sparse Matrix-Vector Prod. & CG-CUBLAS



	scalar-kernel			vector-kernel		
Matrix	SPMV	CG	%	SPMV	CG	%
AUDI	135,63	135,98	99,7	8,83	9,22	95,5
BMW	17,97	18,07	99,4	1,24	1,34	91,7
CRANK	24,82	24,87	99,8	1,16	1,22	94,7
F1	47,93	48,1	99,6	3,61	3,78	95,2
INLINE	63,88	64,21	99,4	4,65	4,92	94,1
LDOOR	78,72	79,15	99,4	7,26	7,72	93,6
A100	2,11	2,54	79,9	5,85	6,29	92,5
A126	4,33	5,14	81,2	11,7	12,52	92,9
A159	8,93	10,49	82,5	23,42	25,01	93,2
A200	17,51	20,56	82,6	46,56	49,62	93,4
A252	35,53	41,66	82,7	93,65	99,79	93,4

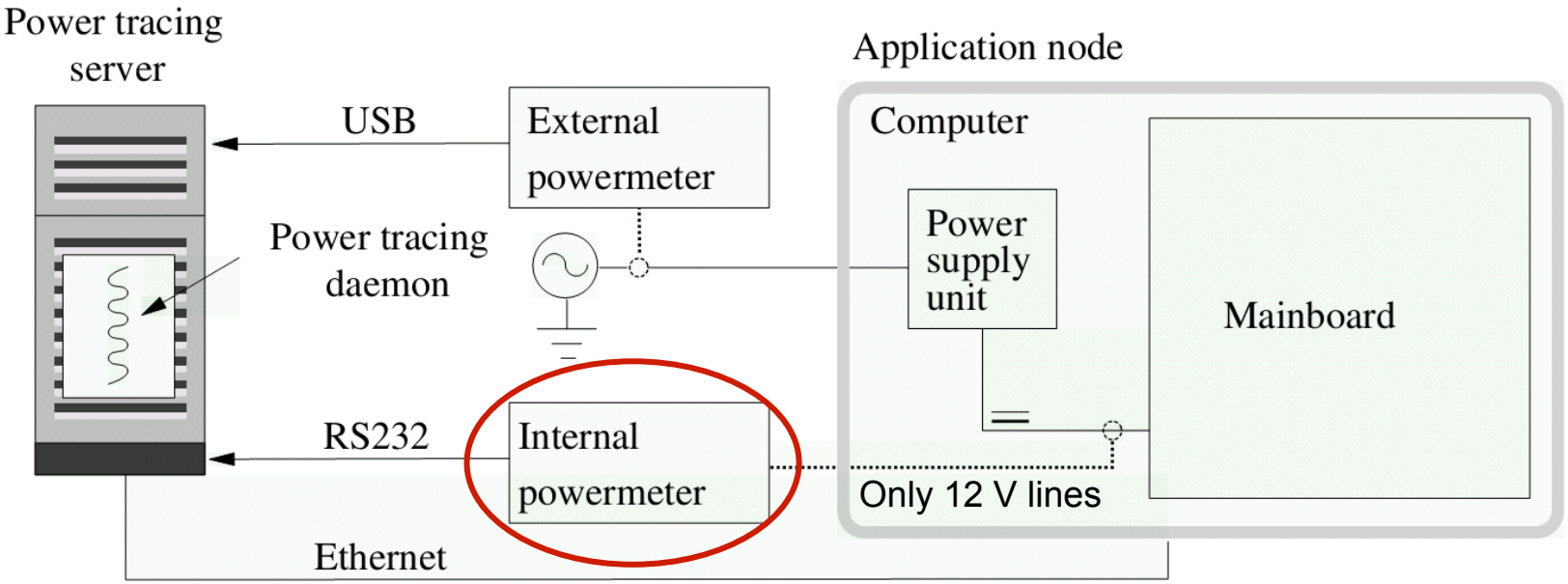
CG-CUBLAS & Energy Consumption

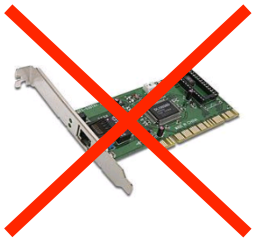


- CUDA programs interleave code executed in CPU threads and CUDA kernels executed in the GPU.
- Some CUBLAS routines are asynchronous, but data dependencies forces to suspend CPU thread.
- Using the polling operation mode of CUDA, the CPU thread wastes power when waiting for the termination of the kernel.
- In the blocking operation mode, the CPU thread is suspended (going to an inactive power-friendly C-state) when waiting for the GPU to finish work.

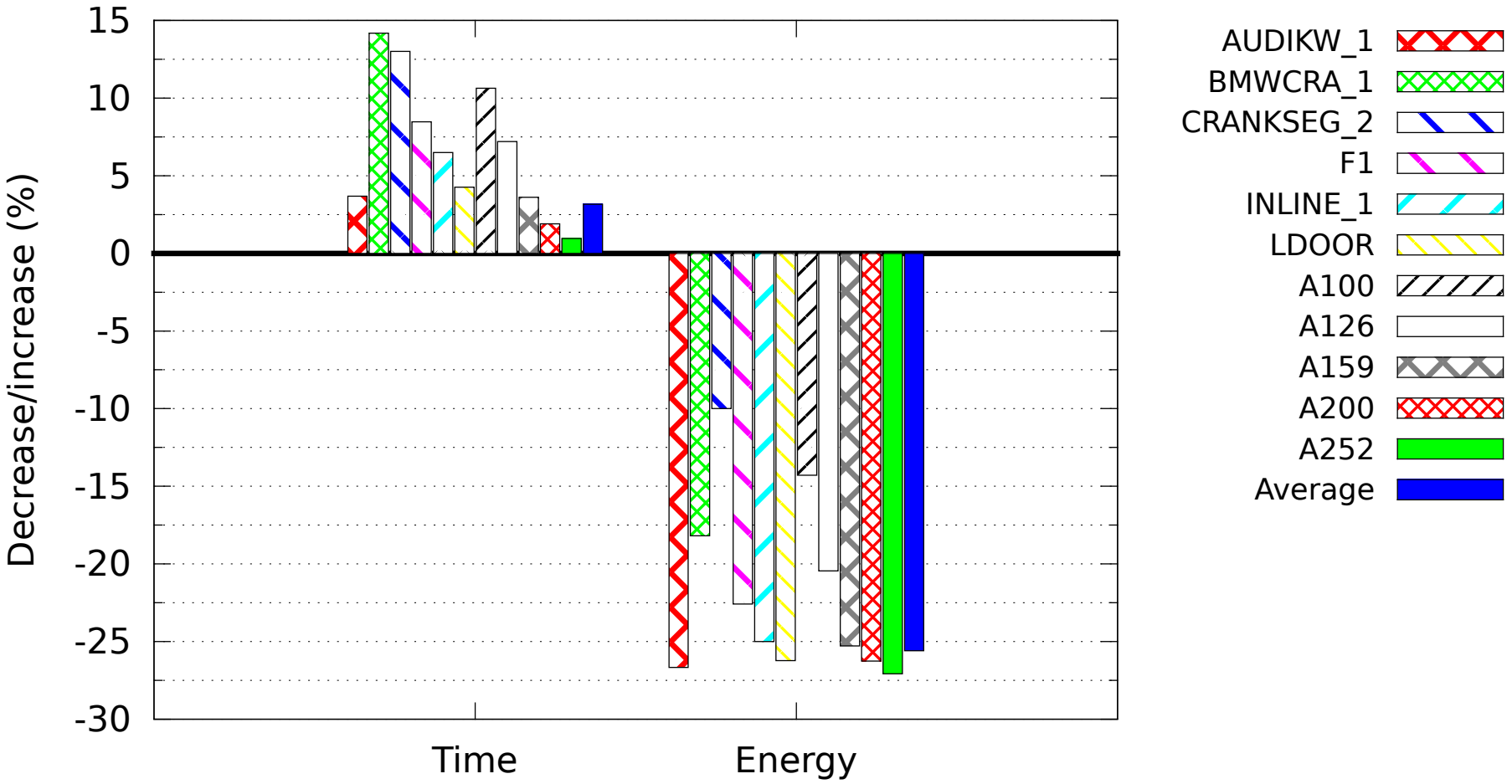
CG-CUBLAS & Energy Consumption

- DC powermeter with sampling freq. = 505 Hz
  - LEM HXS 20-NP transductors with PIC microcontroller
  - RS232 serial port

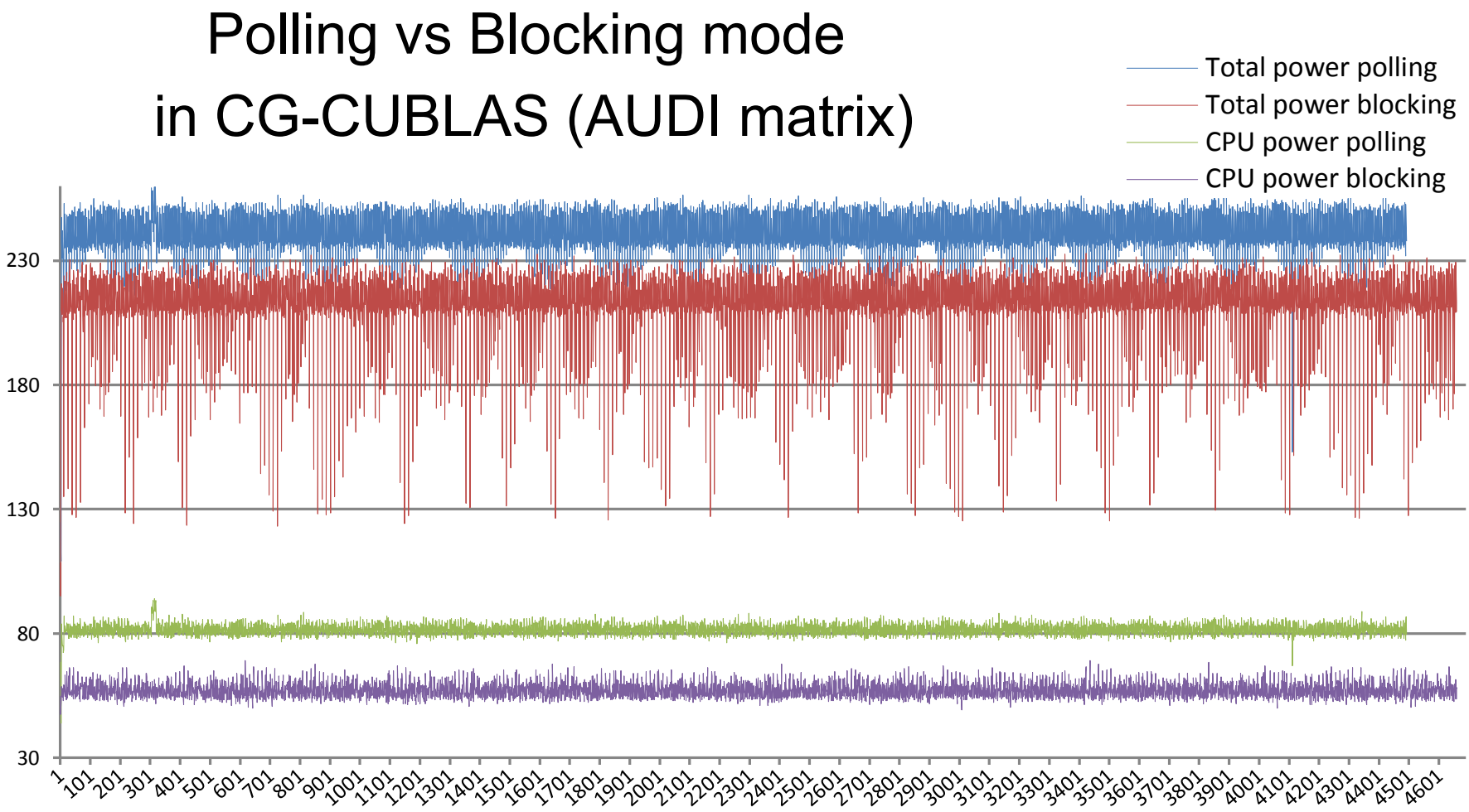




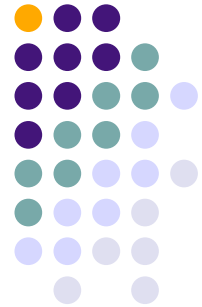
CG-CUBLAS & Energy Consumption



CG-CUBLAS & Energy Consumption

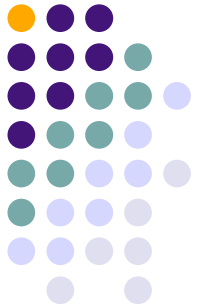


CG-CUDA Implementation



- The CG-CUBLAS loop includes many kernel calls and frequent data transfers (CPU←→GPU).
- An alternative is implement specific CUDA kernels to solve the different operations of the CG.
  - Maintaining or improving the performances of CUBLAS routines.
- The CG-CUDA maps vectors and scalar in the GPU memory:
  - Avoiding synchronizations and data transfers.
  - Allowing to reduce the number of checking steps.
  - Allowing the kernel agregation.

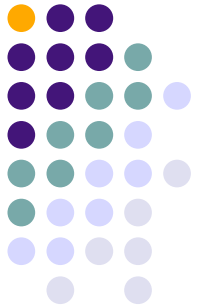
CG-CUDA Implementation basic implementation



```
k = 0;
while( ( k < maxiter ) && ( res > epsilon ) ) {
  SSpMV <<<Gs,Bs>>>( n, rowA, colA, valA, d, z );     // G z := A * d
  cudaSdot( n, d, z, tmp, tmp2 );                     // G tmp := d' * z
  cudaSdiv <<<1,  1>>>( beta, tmp, rho );             // G rho := beta / tmp
  cudaSaxpyadd <<< Gs, Bs>>>( n, rho, d, x );         // G x := x + rho * d
  cudaSaxpysub <<< Gs, Bs>>>( n, rho, z, r );         // G r := r - rho * z
  cudaMemcpy( gamma, beta, sizeof_float,
                  cudaMemcpyDeviceToDevice );         // G gamma := beta
  cudaSdot( n, r, r, tmp, tmp2 );                     // G tmp := r' * r
  cudaMemcpy( beta, tmp, sizeof_float,
                  cudaMemcpyDeviceToDevice );         // G beta  := tmp
  cudaSdiv <<<1,  1>>>( beta, gamma, alpha );         // G alpha := beta/gamma
  cudaSxpayadd <<< Gs, Bs>>>( n, alpha, r, d );       // G d := r + alpha * d
  cudaMemcpy( &res, beta, sizeof_float,
                  cudaMemcpyDeviceToHost );           // C res := || r ||_2^2
  res = sqrt( res );                                  // C res := sqrt(res)
  k++;
}
```

1 transfer/synchronization each iteration

CG-CUDA Implementation reducing the checking steps

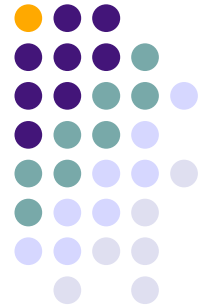


```
k = 0;
while( ( k < maxiter ) && ( res > epsilon ) ){
  for ( i = 0; i < s; i++ ) {
    SSpMV <<<Gs,Bs>>>( n, rowA, colA, valA, d, z )        // G z := A * d
    cudaSdot( n, d, z, tmp, tmp2 );                       // G tmp := d' * z
    cudaSdiv     <<<1,  1>>>( beta, tmp, rho );           // G rho := beta / tmp
    cudaSaxpyadd <<< Gs, Bs>>>( n, rho, d, x );           // G x := x + rho * d
    cudaSaxpysub <<< Gs, Bs>>>( n, rho, z, r );           // G r := r - rho * z
    cudaMemcpy( gamma, beta, sizeof_float,
                  cudaMemcpyDeviceToDevice );             // G gamma := beta
    cudaSdot( n, r, r, tmp, tmp2 );                       // G tmp := r' * r
    cudaMemcpy( beta, tmp, sizeof_float,
                  cudaMemcpyDeviceToDevice );             // G beta  := tmp
    cudaSdiv     <<<1,  1>>>( beta, gamma, alpha );       // G alpha := beta/gamma
    cudaSxpayadd <<< Gs, Bs>>>( n, alpha, r, d );         // G d := r + alpha * d
  }
  cudaMemcpy( &res, beta, sizeof_float,
                  cudaMemcpyDeviceToHost );               // C res := || r ||_2^2
  res = sqrt( res );                                      // C res := sqrt(res)
  k += s;
}
```

1 transfer/synchronization each s iterations

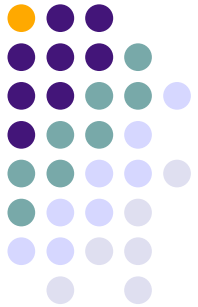
We save (1 – 1/s) transfers/synchronizations

CG-CUDA Implementation
axpy-like kernels and cudaSdot routine



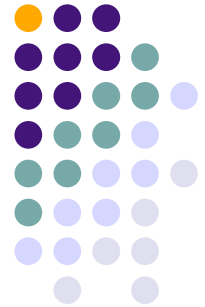
- The axpy-like kernels are very simple, and it is possible to define a specific xpay kernel.

CG-CUDA Implementation
axpy-like kernels



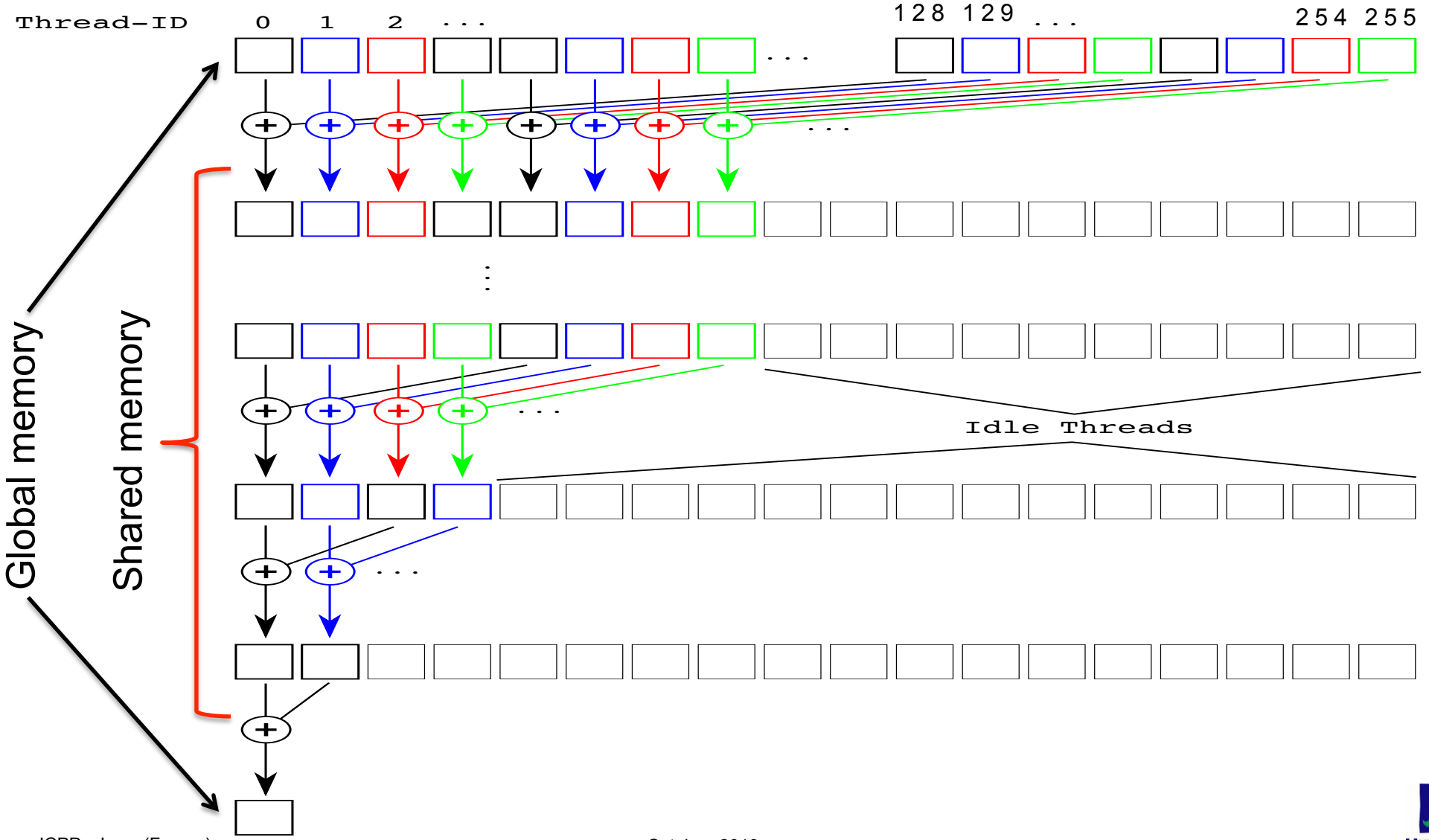
```
__global__ void cudaSsaxpyadd( unsigned int n, float *a, float *x, float *y ) {
  unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

  if( i < n )
    y[ i ] =  a[0] * x[ i ] + y[ i ];
}

__global__ void cudaSsaxpysub( unsigned int n, float *a, float *x, float *y ) {
  unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

  if( i < n )
    y[ i ] = -a[0] * x[ i ] + y[ i ];
}

__global__ void cudaSxpayadd ( unsigned int n, float *a, float *x, float *y ) {
  unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

  if( i < n )
    y[ i ] =  x[ i ] + a[0] * y[ i ];
}
```

CG-CUDA Implementation
axpy-like kernels and cudaSdot routine

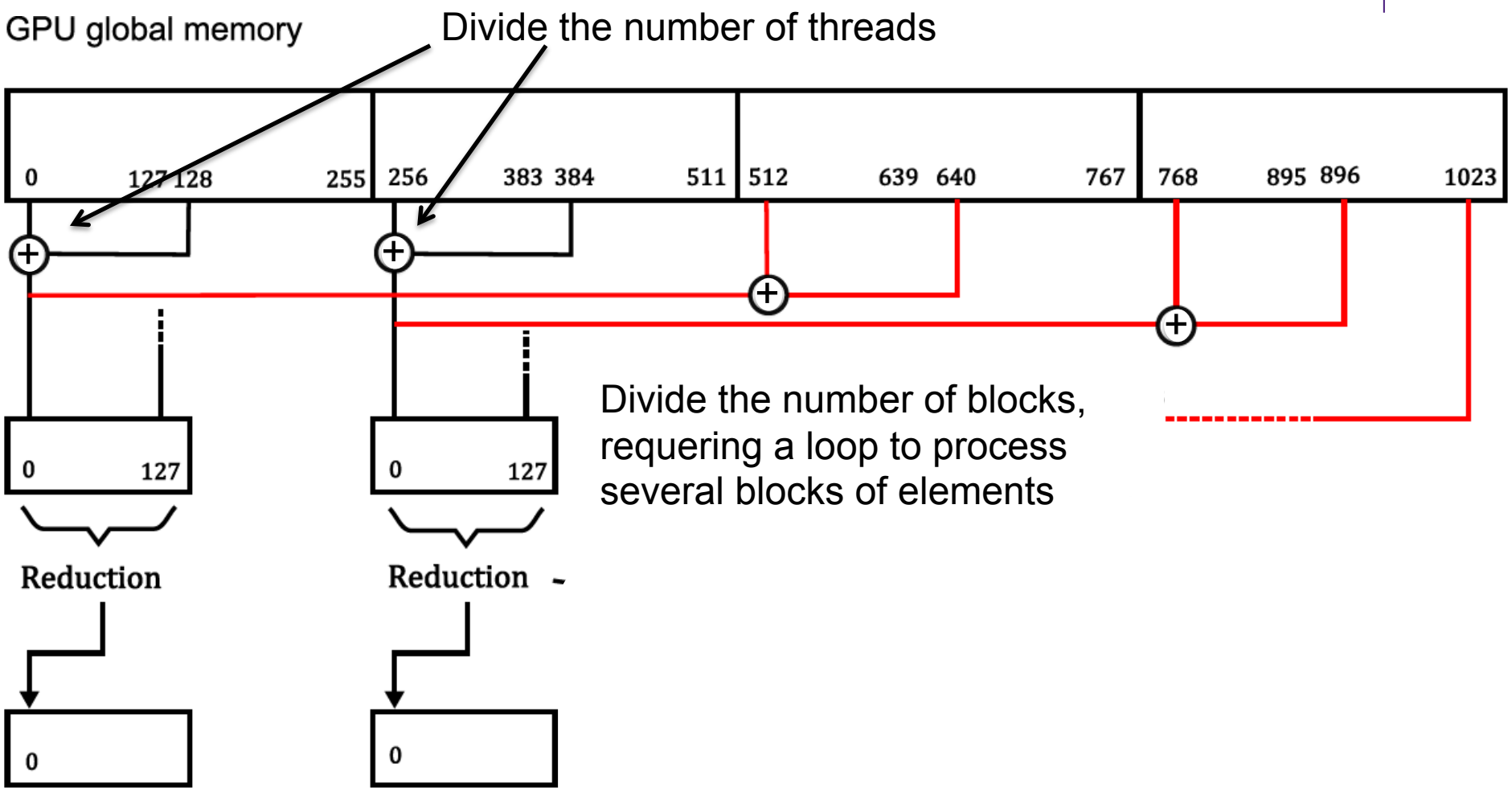


- The axpy-like kernels are very simple, and it is possible to define a specific xpay kernel.
- The dot operation requires a iterative computation,
  - First, the component-wise multiplication is computed.
  - Then, a iterative reduction is applied, in which the size of the vector is divided by the block size in each step.
  - The reduction steps use the shared memory of the GPU

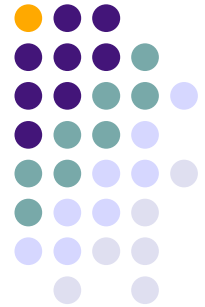
CG-CUDA Implementation
reduction step in shared memory



CG-CUDA Implementation
modified reduction step in shared mem.

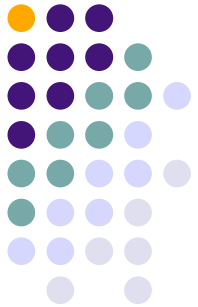


CG-CUDA Implementation
axpy-like kernels and cudaSdot routine



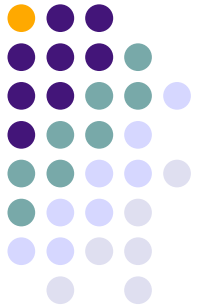
- The axpy-like kernels are very simple, and it is possible to define a specific xpay kernel.
- The dot operation requires a iterative computation,
  - First, the component-wise multiplication is computed.
  - Then, a iterative reduction is applied, in which the size of the vector is divided by the block size in each step.
  - The reduction steps use the shared memory of the GPU
- The cudaSdot routine implements the iteration,
  - cudaSdot_1 kernel computes the component-wise multiplication and the first reduction step
  - cudaSdot_2 kernel computes a modified reduction step

CG-CUDA Implementation
cudaSdot routine



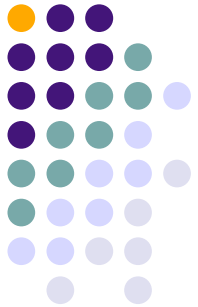
```
__global__ void cudaSdot (int Gs, int Bs, int Ms, int n, float x[],
                          float y[], float vtmp[] , float vtmp2[] ) {
  float *aux1 = vtmp, *aux2 = vtmp2;
  int b = 1, Gs_next, Bs_2 = Bs / 2, Ms_2 = Ms / 2;        // reduce threads

  Gs_next = ( unsigned int ) ceil( ( float ) Gs / Bs );
  cudaSdot_1 <<< Gs_next, Bs, Ms >>> ( n, x, y, aux1 );
  Gs = Gs_next;

  while( Gs > 1 ){
    Gs_next = ( unsigned int ) ceil( ( float ) Gs / Bs );
    if (Gs_next > 1) Gs_next = Gs_next / 2;                // reduce blocks
    cudaSdot_2 <<< Gs_next, Bs_2, Ms_2 >>> ( Gs, aux1, aux2 );
    Gs = Gs_next;
    b = 1 - b;
    if( b ){ aux1 = vtmp; aux2 = vtmp2; }
    else   { aux2 = vtmp; aux1 = vtmp2; }
  }
  if( b == 0 )
    cudaMemcpy( vtmp, aux1, sizeof( float ), cudaMemcpyDeviceToDevice );
}
```

CG-CUDA Implementation
cudaSdot routine



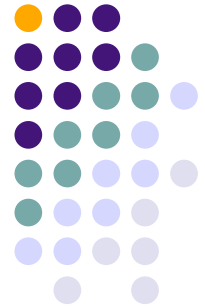
```
__global__ void cudaSdot_1 ( int n, float x[], float y[], float vtmp[] ) {
  uint Idx = threadIdx.x;
  uint i   = blockIdx.x * blockDim.x + Idx;
  extern __shared__ float temp[];

  temp[Idx] = ( i < n ) ? x[i] * y[i] : 0;    __syncthreads();

  if( Idx < 128 ) temp[Idx] += temp[Idx+128]; __syncthreads();

  if( Idx <  64 ) temp[Idx] += temp[Idx+ 64]; __syncthreads();

  if(Idx < 32){
    volatile float *temp2 = temp;
    temp2[Idx] += temp2[Idx+32]; temp2[Idx] += temp2[Idx+16];
    temp2[Idx] += temp2[Idx+ 8]; temp2[Idx] += temp2[Idx+ 4];
    temp2[Idx] += temp2[Idx+ 2]; temp2[Idx] += temp2[Idx+ 1];
  }

  if(Idx == 0) vtmp[blockIdx.x] = temp[0];
}
```

CG-CUDA Implementation cudaSdot routine



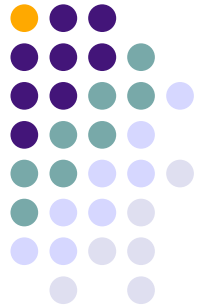
```
__global__ void cudaSdot_2 ( int n, float vtmp[], float vtmp2[] ) {
  extern __shared__ float temp[];
  uint Idx = threadIdx.x, blockSize = blockDim.x;
  uint i = blockIdx.x * ( blockSize * 2 ) + Idx;
  uint gridSize = blockSize * 2 * gridDim.x;

  temp[ Idx ] = 0;
  while ( i < n ) {
    temp[ Idx ] += vtmp[i];
    temp[ Idx ] += ( i + blockSize < n )? vtmp[ i + blockSize ] : 0;
    i += gridSize;
  }
  __syncthreads();
  if( Idx <  64 ) temp[Idx] += temp[Idx+ 64]; __syncthreads();
  if(Idx < 32){
    volatile float *temp2 = temp;
    temp2[Idx] += temp2[Idx+32]; temp2[Idx] += temp2[Idx+16];
    temp2[Idx] += temp2[Idx+ 8]; temp2[Idx] += temp2[Idx+ 4];
    temp2[Idx] += temp2[Idx+ 2]; temp2[Idx] += temp2[Idx+ 1];
  }

  if(Idx == 0) vtmp2[blockIdx.x] = temp[0];
}
```

Merging kernel on CG-CUDA



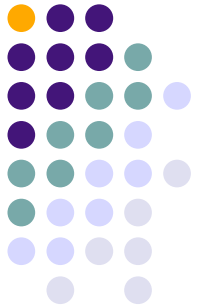
- Merge CUDA kernels is a good alternative to improve the performances.
- The CUDA data-parallel programming model force to GPU kernels must contain only operations that are independent or can be executed consecutively.
- Our objective is to gather vector operations that can be executed independently or consecutively on a subvector without synchronizing with the other parts.

Merging kernel on CG-CUDA



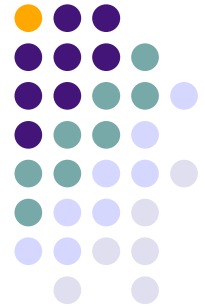
```
k = 0;
while( ( k < maxiter ) && ( res > epsilon ) ) {
  SSpMV <<<Gs,Bs>>>( n, rowA, colA, valA, d, z );          //scalar_fusion_1
  cudaSdot( n, d, z, tmp,tmp2 );
  cudaSdiv <<<1,  1>>>( beta, tmp, rho );                  //fusion_2
  cudaSaxpyadd <<< Gs, Bs>>>( n, rho, d, x );
  cudaSaxpysub <<< Gs, Bs>>>( n, rho, z, r );              //fusion_3
  cudaMemcpy( gamma, beta, sizeof_float,
                  cudaMemcpyDeviceToDevice );
  cudaSdot( n, r, r, tmp, tmp2 );                          //fusion_4
  cudaMemcpy( beta, tmp, sizeof_float,
                  cudaMemcpyDeviceToDevice );
  cudaSdiv <<<1,  1>>>( beta, gamma, alpha );              //fusion_5
  cudaSxpayadd <<< Gs, Bs>>>( n, alpha, r, d );
  cudaMemcpy( &res, beta, sizeof_float,
                  cudaMemcpyDeviceToHost );          // res := || r ||_2^2
  res = sqrt( res );                                 // res := sqrt(res)
  k++;
}
```

Merging kernel on CG-CUDA
fusion_3 kernel



```
__global__ void fusion_3 (int n, float rho[], float d[], float x[],
                          float z[], float r[], float vtmp[]  ) {
  uint Idx = threadIdx.x,  i = blockIdx.x * blockDim.x + Idx;
  extern __shared__ float temp[];

  if(i < n){
    x[i] =  rho[0] * d[i] + x[i];
    r[i] = -rho[0] * z[i] + r[i];
  }
  __syncthreads();
  temp[Idx] = ( i < n ) ? r[i] * r[i] : 0;     __syncthreads();
  if( Idx < 128 ) temp[Idx] += temp[Idx+128]; __syncthreads();
  if( Idx <  64 ) temp[Idx] += temp[Idx+ 64]; __syncthreads();
  if(Idx < 32){
    volatile float *temp2 = temp;
    temp2[Idx] += temp2[Idx+32]; temp2[Idx] += temp2[Idx+16];
    temp2[Idx] += temp2[Idx+ 8]; temp2[Idx] += temp2[Idx+ 4];
    temp2[Idx] += temp2[Idx+ 2]; temp2[Idx] += temp2[Idx+ 1];
  }

  if(Idx == 0) vtmp[blockIdx.x] = temp[0];
}
```

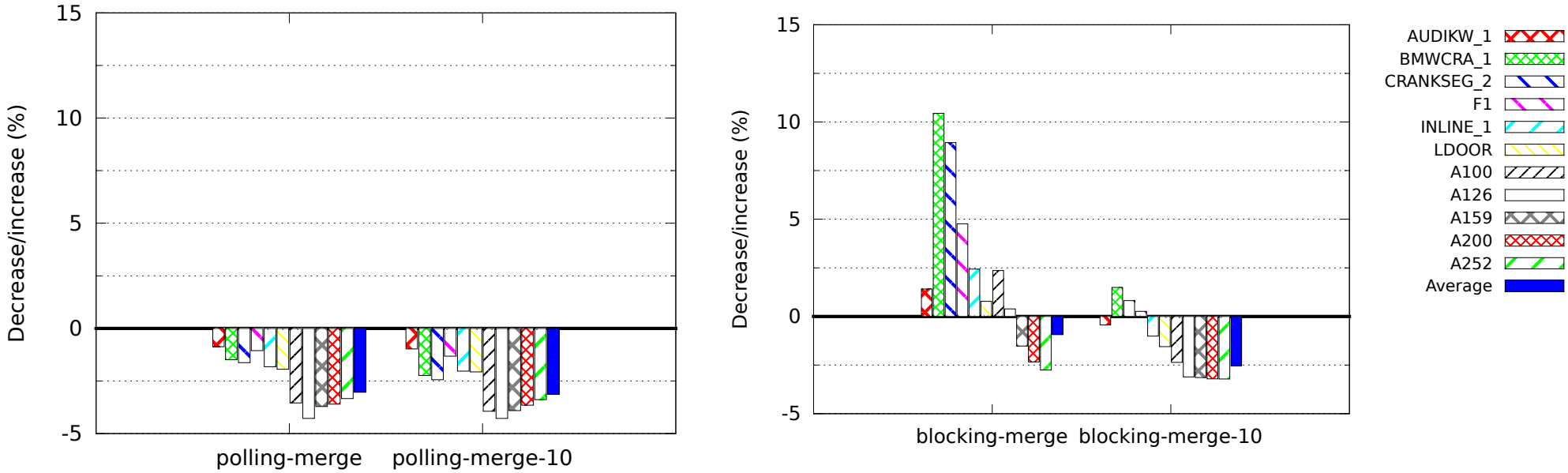
Experimental Results



- In the analysis of the results we consider the following algorithms:
  - CG-CUBLAS.
  - CG-CUDA-Merge
  - CG-CUDA-Merge with reducing cheking steps (s = 10).
- We consider the runtime and the energy consumption for the polling and blocking modes.
- Usually, the polling mode execution of the CG-CUBLAS is the reference value.

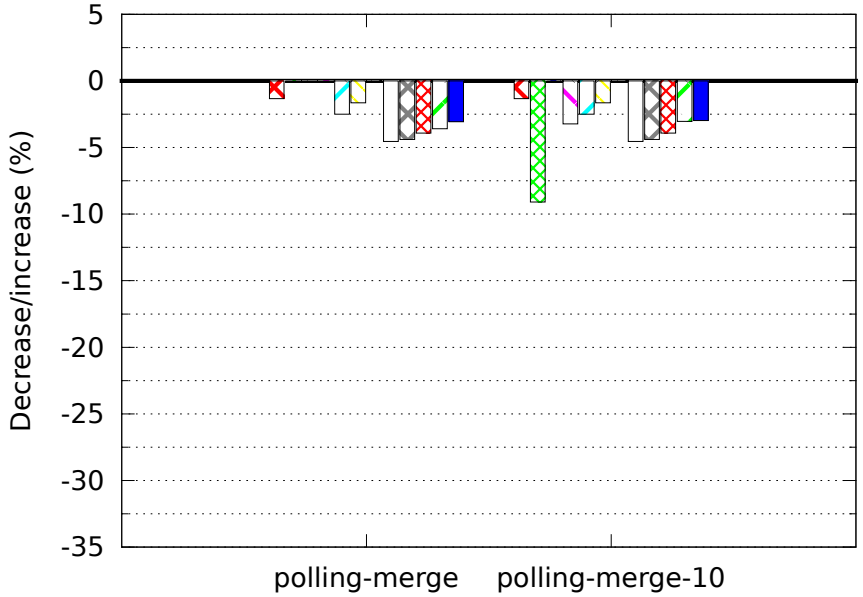
Experimental Results

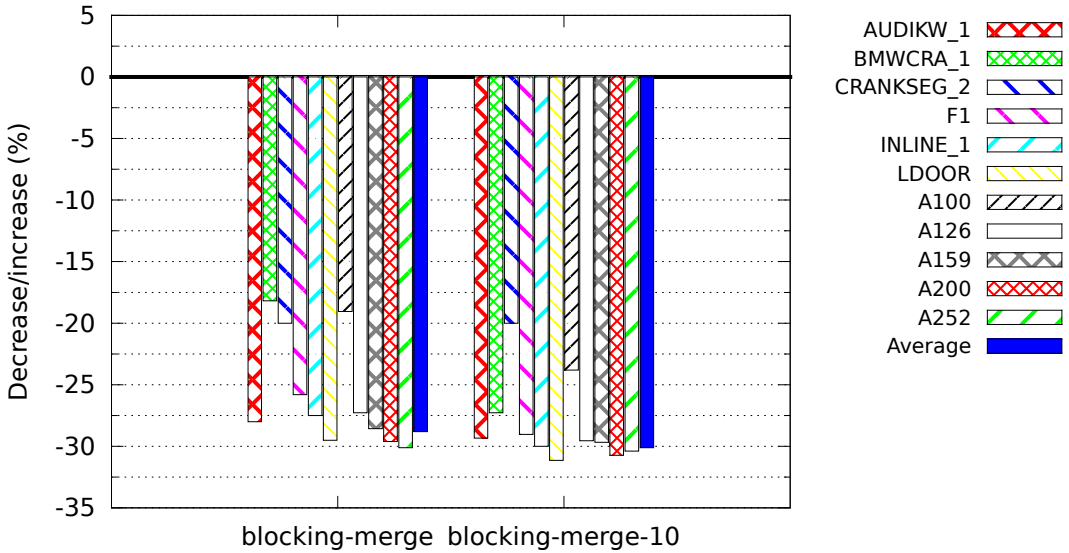
Analysis of the Computational Time



Experimental Results

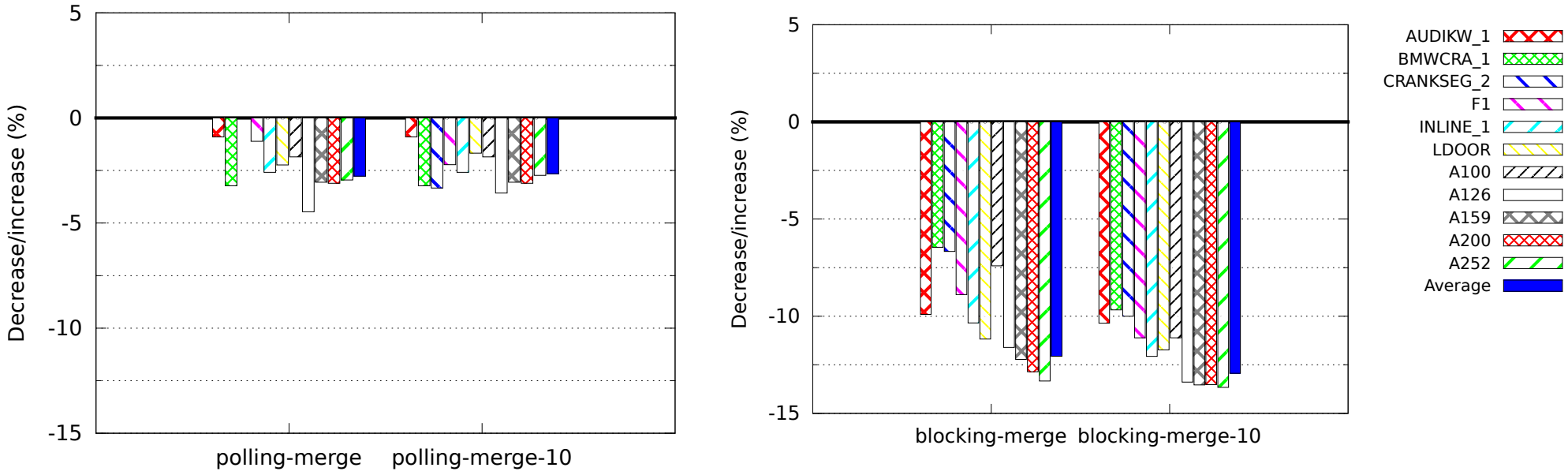
Analysis of the CPU Energy Consumption



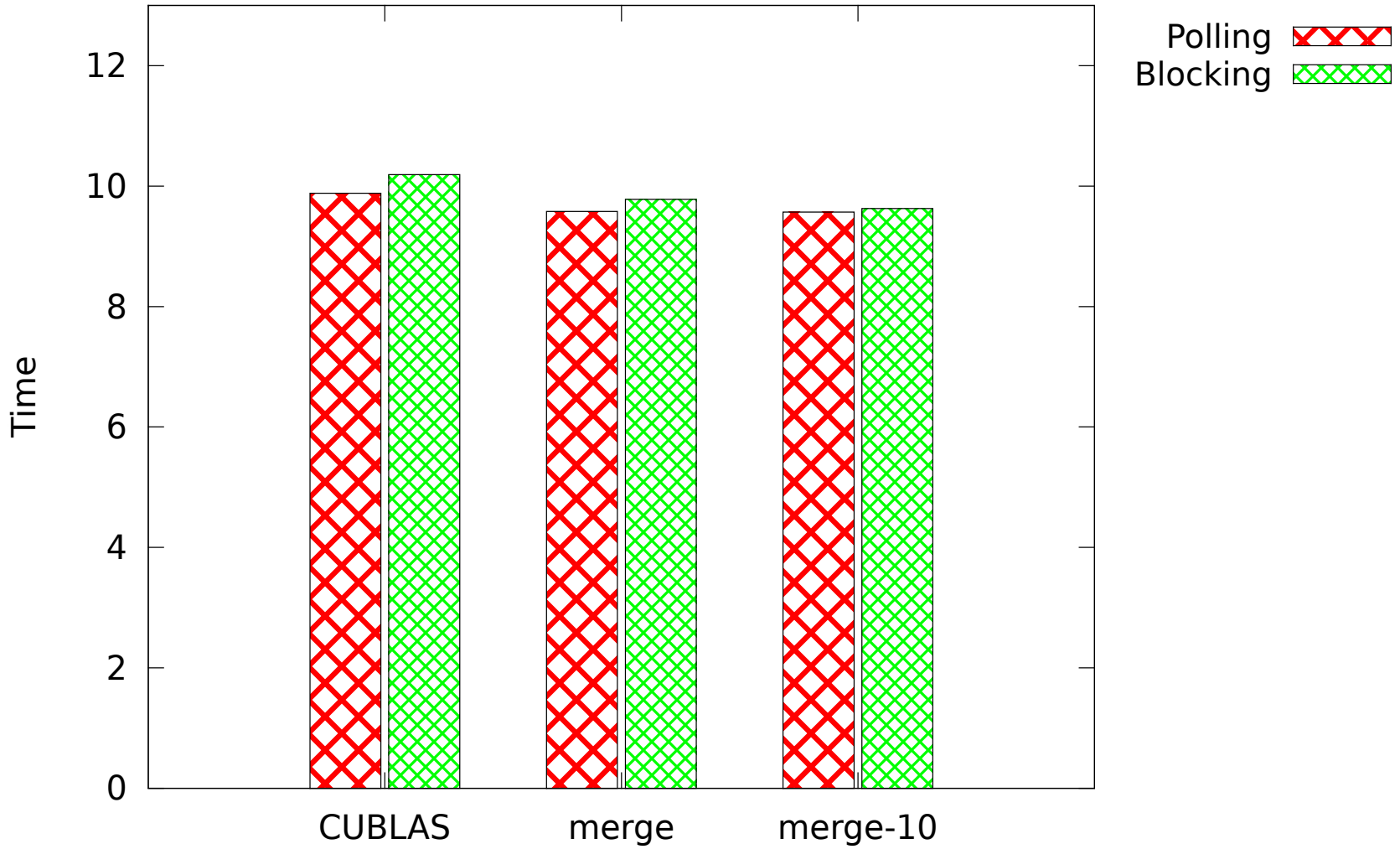


Experimental Results

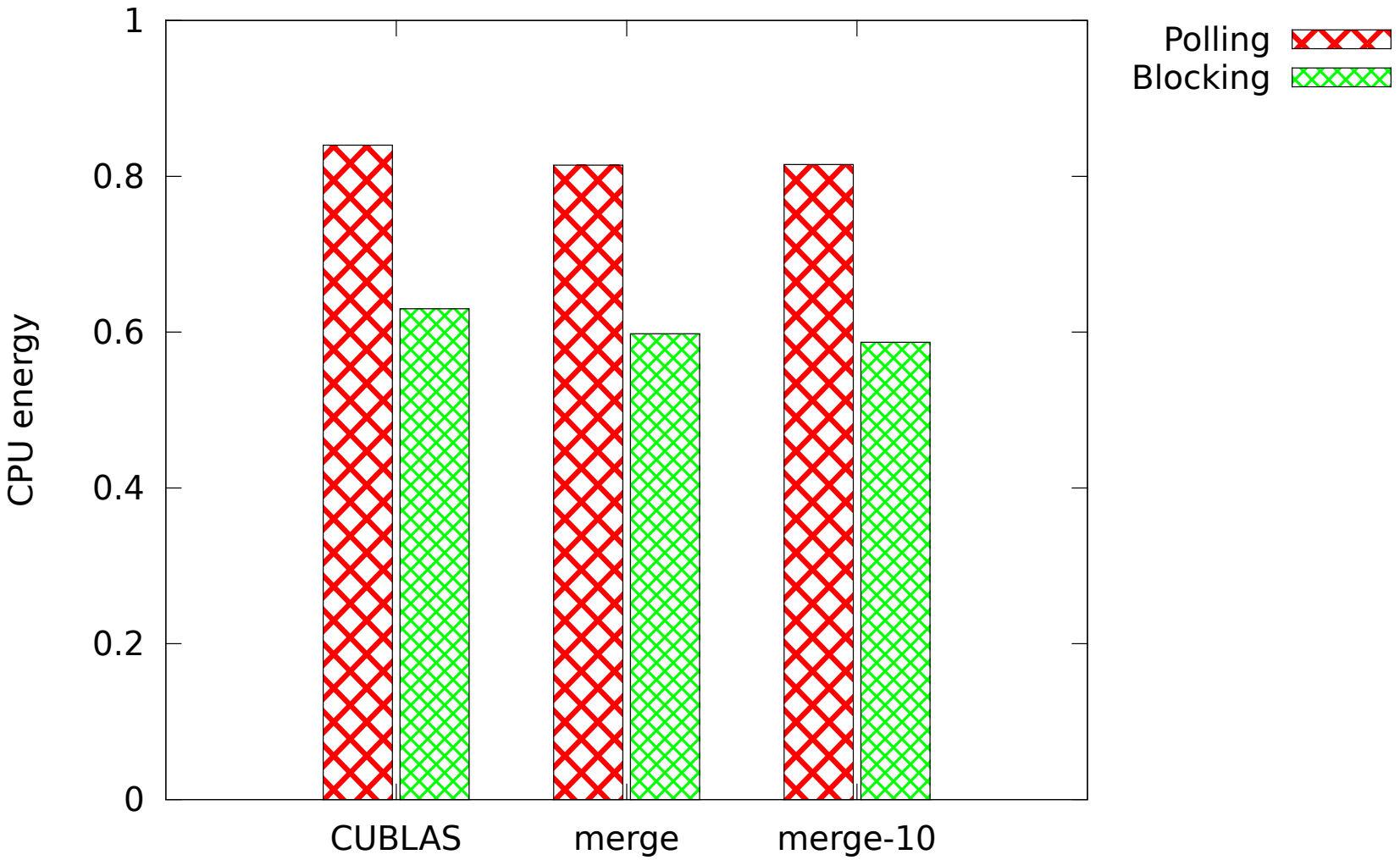
Analysis of the Total Energy Consumption



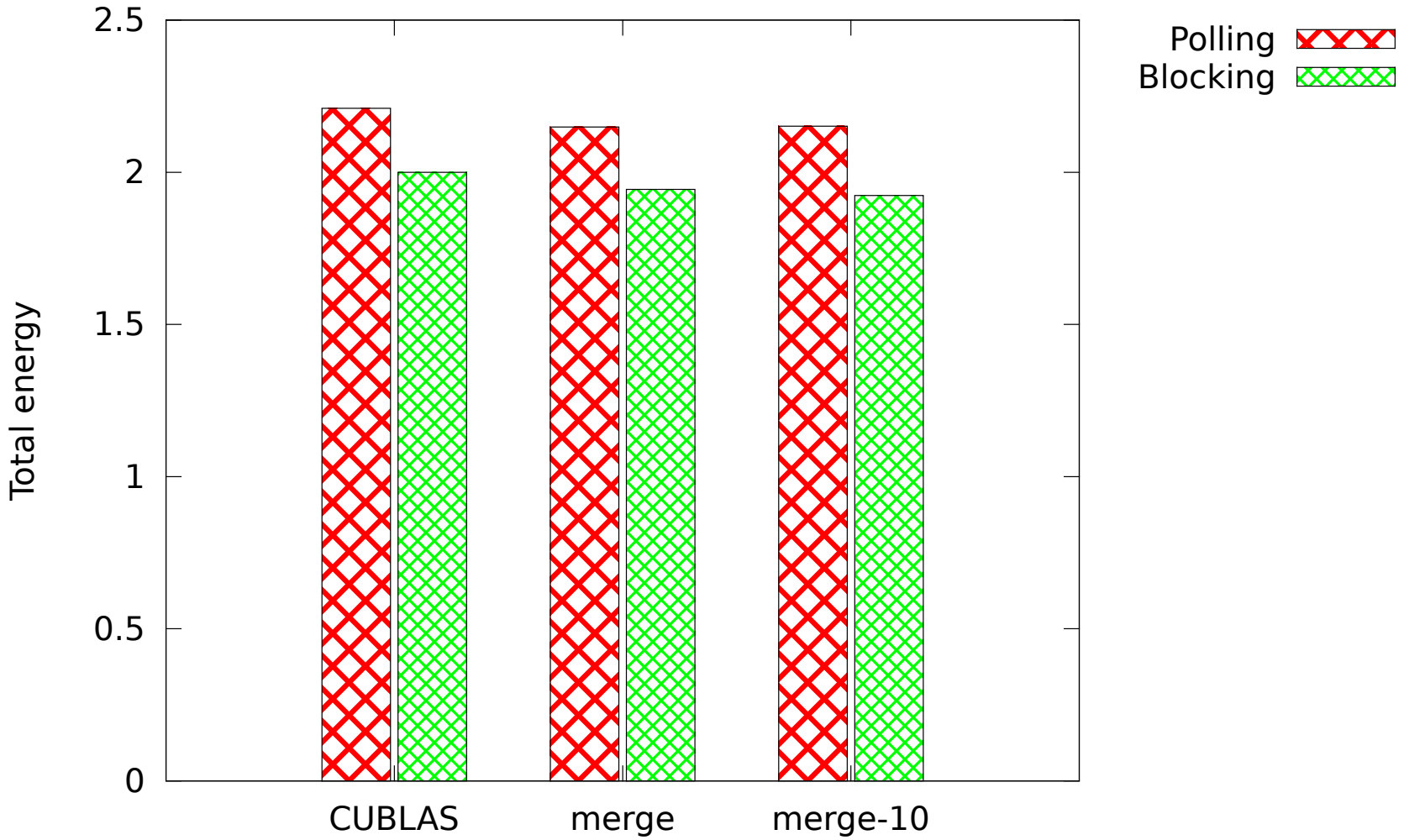
Experimental Results



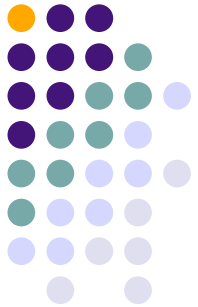
Experimental Results



Experimental Results



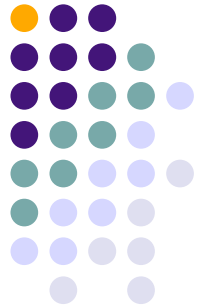
Conclusions & Future work



- We have developed on GPU a energy-aware implementation of the CG:
  - Reformulating the CG and using energy saving mechanisms.
  - Improving around 2,5% the runtime.
  - Saving around 30% the CPU energy consumption.

- The future research should focus on detecting repetitive patterns to rearrange and merge automatically kernels.

Reformulated Conjugate Gradient for the Energy-Aware Solution of Linear Systems on GPUs



Thanks for your attention!

Any question?