Systematic Fusion of CUDA Kernels for Iterative Sparse Linear System Solvers

José I. ALIAGA

Joaquín PÉREZ

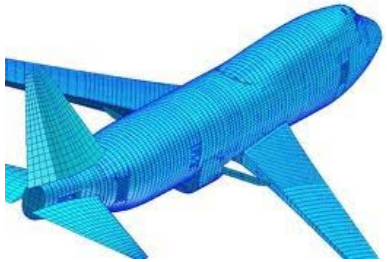
Enrique S. QUINTANA-ORTÍ

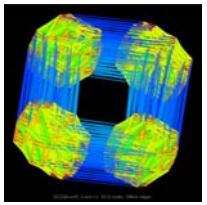
UNIVERSITAT JAUME·I

Motivation
Sparse Linear Systems

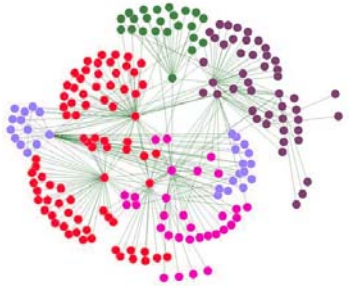
Ubiquitous problem:

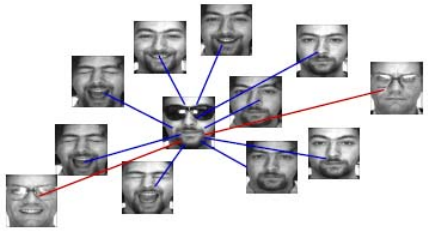
- Boundary value problems and FEM for PDEs
- Quantum chemistry
- Economic modeling





- Web search
- Information retrieval
- Pattern recognition



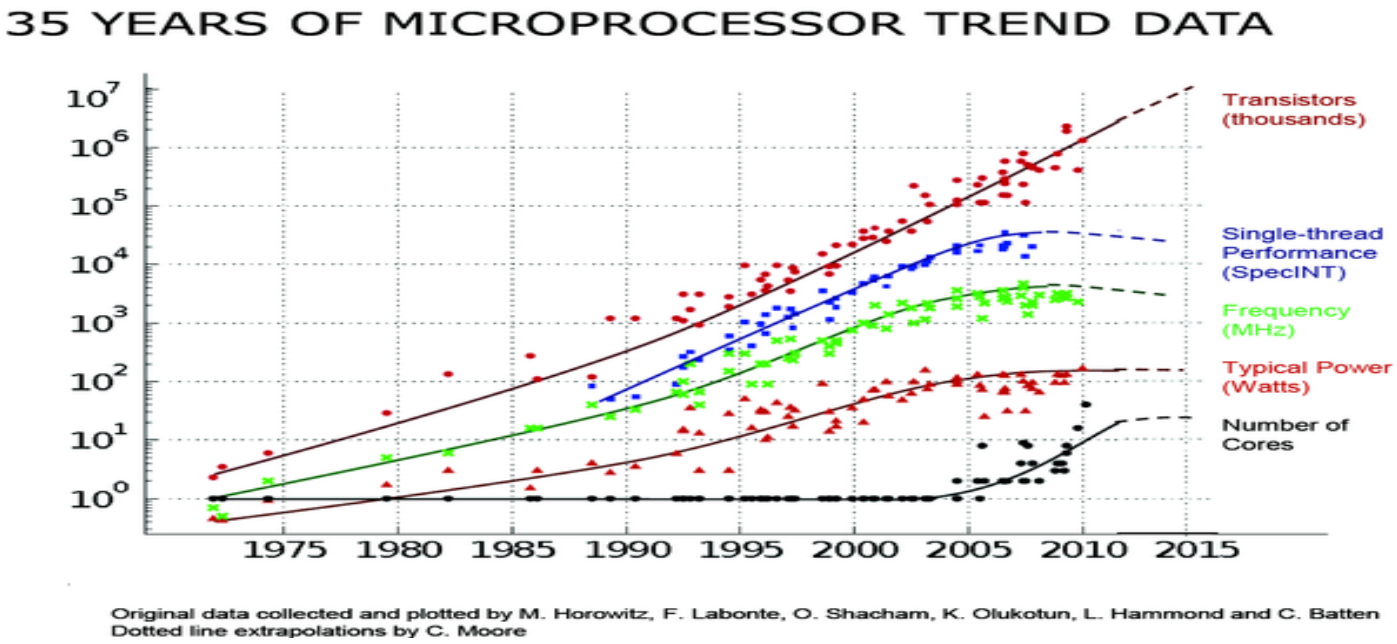


→ Intel MKL, IBM ESSL, NVIDIA cuSPARSE,…

Motivation
Power/Energy/Utilization walls

Accelerator computing

- Moore Law in place
- End of Dennard's scaling
- Dark silicon



Motivation
GPU Computing

Why?

- Moderate Price
- High performance
- Favorable throughput-per-Watt
- Powerful and simple APIs (remember Cell B.E.)
  - OpenACC
  - CUDA
  - OpenCL

June 2015: 90 systems equipped with accelerators

- 52 with NVIDIA GPUs
- 4 with ATI Radeon GPUs
- 4 with NVIDIA GPUs + Intel Xeon Phi

Motivation
Efficiency of sparse solvers on GPUs

Use of libraries (CUBLAS, cuSPARSE) to ease development:

$$
\begin{aligned}
&\textbf{while } (k < maxiter) \ \&\ (res > \varepsilon)\\
&\quad z_k := Ad_k\\
&\quad \rho_k := \frac{\beta_k}{d_k^T z_k}\\
&\quad \gamma_k := \beta_k\\
&\quad x_{k+1} := x_k + \rho_k d_k\\
&\quad r_{k+1} := r_k - \rho_k z_k\\
&\quad \beta_{k+1} := r_{k+1}^T r_{k+1}\\
&\quad \alpha_k := \frac{\beta_{k+1}}{\gamma_k}\\
&\quad d_{k+1} := r_{k+1} + \alpha_k d_k\\
&\quad res := \| r_{k+1} \|_2\\
&\quad k := k + 1\\
&\textbf{end}
\end{aligned}
$$

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

Motivation
Efficiency of sparse solvers on GPUs

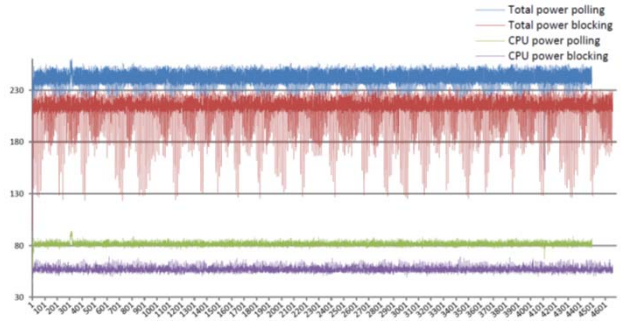
Kernel fusion:

- Improve memory locality
- Reduce overhead of procedure calls
- Richer space for compiler optimizations…

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

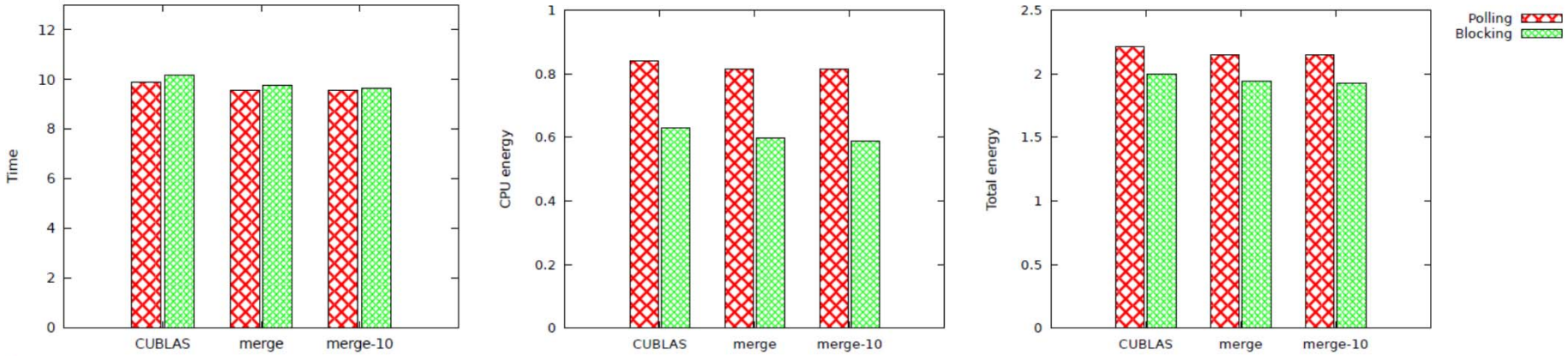
Motivation
Efficiency of sparse solvers on GPUs

Waste of energy for CPU: Hard to leverage energy-saving C-states



Merge CUDA kernels via ad-hoc solvers

"Reformulated Conjugate Gradient for the Energy-Aware Solution of Linear Systems on GPUs"
J. I. Aliaga, J. Pérez, E. S. Quintana-Ortí, H. Anzt
ICPP 2013



Outline

Systematic Fusion of CUDA Kernels for Iterative Solvers

- Iterative solvers for sparse linear systems
  - Identification of kernels
- Fusion
  - Characterization of kernels
- Case studies: BiCG and others
- Experiments
- Concluding remarks

Iterative Solvers for Sparse Linear Systems
Identification of kernels

CG, BiCG, BiCGStab,… with preconditioner

$A \rightarrow M$	Compute Jacobi preconditioner
Initialize $r_0, r_0^*, p_0, p_0^*, x_0, \sigma_0, \tau_0; j := 0$	
while $(\tau_j > \tau_{\max})$	Loop for iterative solver
1. $v_j := Ap_j$	1. SpMV
2. $\alpha_j := \sigma_j / (v_j, p_j^*)$	2. DOT
3. $x_{j+1} := x_j + \alpha_j p_j$	3. AXPY
4. $r_{j+1} := r_j - \alpha_j v_j$	4. AXPY
5. $z_j := M^{-1} r_{j+1}$	5. JPRED (Jacobi preconditioner)
6. $v_j^* := A^T p_j^*$	6. SpMV
7. $r_{j+1}^* := r_j^* - \alpha_j v_j^*$	7. AXPY
8. $z_j^* := M^{-1} r_{j+1}^*$	8. JPRED (Jacobi preconditioner)
9. $\zeta_j := (z_j, r_{j+1}^*)$	9. DOT
10. $\beta_j := \zeta_j / \sigma_j$	10. Scalar op
11. $\sigma_j = \zeta_j$	11. Scalar op
12. $p_{j+1} := z_j + \beta_j p_j$	12. XPAY (AXPY-like)
13. $p_{j+1}^* := z_j^* + \beta_j p_j^*$	13. XPAY (AXPY-like)
14. $\tau_{j+1} := \| r_{j+1} \|_2$	14. Vector 2-norm (DOT + sqrt)
$j := j + 1$	
endwhile	

SpMV (different formats)
Preconditioner appl.
BLAS-1: AXPY-like, DOT

Fusion
Characterization of kernels

Mapped vs unmapped:

- A kernel K performs a *mapped access* to a vector v if each thread of K accesses one element of v, independently of other threads, and the global access is coalesced

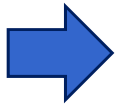
Operation		Input vector(s) x	Input vector(s) y/M^{-1}	Output vector y
AXPY	$y := \alpha x + y$	mapped	mapped	mapped
XPAY	$y := \alpha y + x$	mapped	mapped	mapped
JPRED	$y := M^{-1}x$	mapped	mapped	mapped
DOT	$\alpha := x^T y = (x, y)$	mapped	mapped	unmapped
SPMV vector CSR	$y := Ax$	unmapped	–	unmapped
SPMV scalar CSR	$y := Ax$	unmapped	–	mapped
SPMV ELL	$y := Ax$	unmapped	–	mapped

Fusion
Conditions

Merge of K_1 and K_2 possible?

- K_1 and K_2 are independent
- If K_1 produces an output v and K_2 takes this vector as an input (RAW dependency):

K_1 and K_2 perform mapped accesses to the output/input vector v



1. Both kernels apply the same mapping of threads to the vector elements shared (exchanged) via registers
2. Both kernels apply the same mapping of thread blocks to the vector elements shared (exchanged) via shared memory
3. A global barrier is not necessary between the two kernels

Fusion
Application

K_1 and K_2 perform mapped accesses to the input/output vector v

Operation		Input vector(s) x	Input vector(s) y/M^{-1}	Output vector y
AXPY	$y := \alpha x + y$	mapped	mapped	mapped
XPAY	$y := \alpha y + x$	mapped	mapped	mapped
JPRED	$y := M^{-1}x$	mapped	mapped	mapped
DOT	$\alpha := x^T y = (x, y)$	mapped	mapped	unmapped
SPMV vector CSR	$y := Ax$	unmapped	–	unmapped
SPMV scalar CSR	$y := Ax$	unmapped	–	mapped
SPMV ELL	$y := Ax$	unmapped	–	mapped

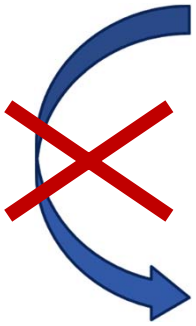
Fusion
Application

K_1 and K_2 perform mapped accesses to the input/output vector v

Operation		Input vector(s) x	Input vector(s) y/M^{-1}	Output vector y
AXPY	$y := \alpha x + y$	mapped	mapped	mapped
XPAY	$y := \alpha y + x$	mapped	mapped	mapped
JPRED	$y := M^{-1}x$	mapped	mapped	mapped
DOT	$\alpha := x^T y = (x, y)$	mapped	mapped	unmapped
SPMV vector CSR	$y := Ax$	unmapped	–	unmapped
SPMV scalar CSR	$y := Ax$	unmapped	–	mapped
SPMV ELL	$y := Ax$	unmapped	–	mapped

Fusion
Application

K_1 and K_2 perform mapped accesses to the input/output vector v



Operation		Input vector(s) x	Input vector(s) y/M^{-1}	Output vector y
AXPY	$y := \alpha x + y$	mapped	mapped	mapped
XPAY	$y := \alpha y + x$	mapped	mapped	mapped
JPRED	$y := M^{-1}x$	mapped	mapped	mapped
DOT	$\alpha := x^T y = (x, y)$	mapped	mapped	unmapped
SPMV vector CSR	$y := Ax$	unmapped	–	unmapped
SPMV scalar CSR	$y := Ax$	unmapped	–	mapped
SPMV ELL	$y := Ax$	unmapped	–	mapped

Fusion
Application

K_1 and K_2 perform mapped accesses to the input/output vector v

Operation		Input vector(s) x	Input vector(s) y/M^{-1}	Output vector y
AXPY	$y := \alpha x + y$	mapped	mapped	mapped
XPAY	$y := \alpha y + x$	mapped	mapped	mapped
JPRED	$y := M^{-1}x$	mapped	mapped	mapped
DOT	$\alpha := x^T y = (x, y)$	mapped	mapped	unmapped
SPMV vector CSR	$y := Ax$	unmapped	–	unmapped
SPMV scalar CSR	$y := Ax$	unmapped	–	mapped
SPMV ELL	$y := Ax$	unmapped	–	mapped

Fusion
Application

K_1 and K_2 perform mapped accesses to the input/output vector v

Operation		Input vector(s) x	Input vector(s) y/M^{-1}	Output vector y
AXPY	$y := \alpha x + y$	mapped	mapped	mapped
XPAY	$y := \alpha y + x$	mapped	mapped	mapped
JPRED	$y := M^{-1}x$	mapped	mapped	mapped
DOT	$\alpha := x^T y = (x, y)$	mapped	mapped	unmapped
SPMV vector CSR	$y := Ax$	unmapped	–	unmapped
SPMV scalar CSR	$y := Ax$	unmapped	–	mapped
SPMV ELL	$y := Ax$	unmapped	–	mapped

?

Fusion
Application

Two-stage DOT:

- DOT_{ini}
  - CUDA kernel
  - Costly element-wise products and reduction within a single thread block
  - Mapped input → Fusible with AXPY, XPAY, JPRED, SpMV (scalar CSR, ELL)
  - Unmapped output
- DOT_{fin}
  - Final reduction into scalar result. Very small cost
  - CUDA routine, implemented as a collection of CUDA kernels
  - Each kernel with mapped input and unmapped output

Fusion
Application

When?

- Reduce as much as possible the number of macro-kernels

How?

- Systematic analysis of pairs of kernels
- In general, merge kernels by “stacking” their codes one after another respecting dependencies
- When merging two DOT_{ini} or two DOT_{fin}, interleave their codes

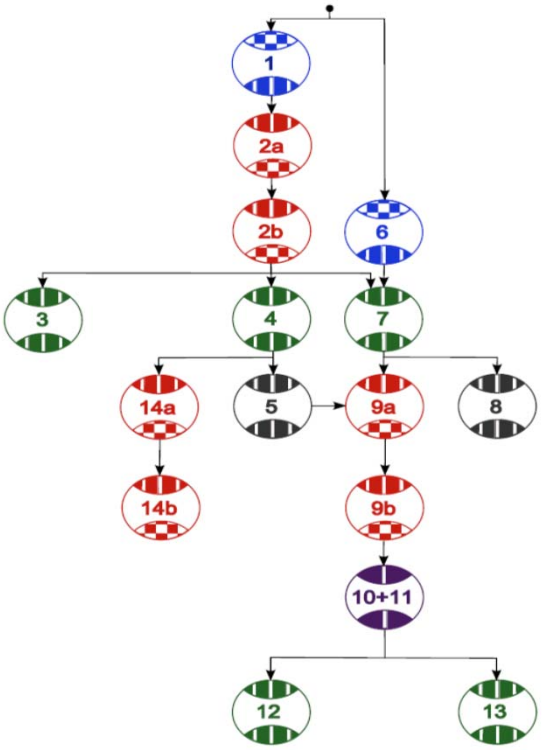
Case Studies
BiCG

Dependencies (two-stage DOT):

$A \rightarrow M$
Initialize $r_0, r_0^*, p_0, p_0^*, x_0, \sigma_0, \tau_0; j := 0$
while $(\tau_j > \tau_{\max})$

1. $v_j := Ap_j$
2. $\alpha_j := \sigma_j / (v_j, p_j^*)$
3. $x_{j+1} := x_j + \alpha_j p_j$
4. $r_{j+1} := r_j - \alpha_j v_j$
5. $z_j := M^{-1} r_{j+1}$
6. $v_j^* := A^T p_j^*$
7. $r_{j+1}^* := r_j^* - \alpha_j v_j^*$
8. $z_j^* := M^{-1} r_{j+1}^*$
9. $\zeta_j := (z_j, r_{j+1}^*)$
10. $\beta_j := \zeta_j / \sigma_j$
11. $\sigma_j = \zeta_j$
12. $p_{j+1} := z_j + \beta_j p_j$
13. $p_{j+1}^* := z_j^* + \beta_j p_j^*$
14. $\tau_{j+1} := \| r_{j+1} \|_2$

$j := j + 1$
endwhile



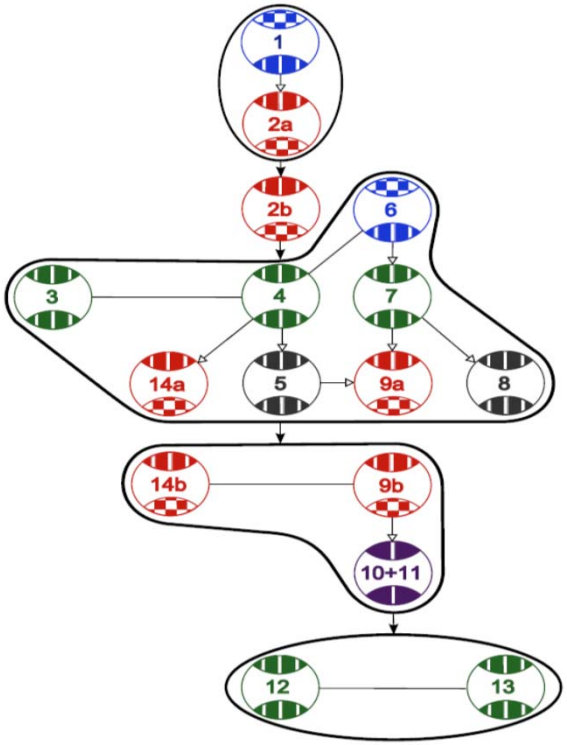
Case Studies
BiCG

Fusions:

$A \rightarrow M$
Initialize $r_0, r_0^*, p_0, p_0^*, x_0, \sigma_0, \tau_0; j := 0$
while $(\tau_j > \tau_{\max})$

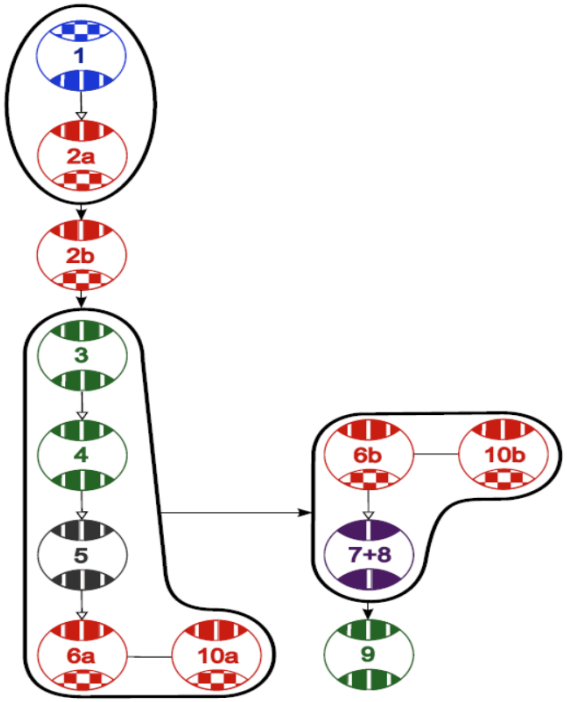
1. $v_j := Ap_j$
2. $\alpha_j := \sigma_j / (v_j, p_j^*)$
3. $x_{j+1} := x_j + \alpha_j p_j$
4. $r_{j+1} := r_j - \alpha_j v_j$
5. $z_j := M^{-1} r_{j+1}$
6. $v_j^* := A^T p_j^*$
7. $r_{j+1}^* := r_j^* - \alpha_j v_j^*$
8. $z_j^* := M^{-1} r_{j+1}^*$
9. $\zeta_j := (z_j, r_{j+1}^*)$
10. $\beta_j := \zeta_j / \sigma_j$
11. $\sigma_j = \zeta_j$
12. $p_{j+1} := z_j + \beta_j p_j$
13. $p_{j+1}^* := z_j^* + \beta_j p_j^*$
14. $\tau_{j+1} := \| r_{j+1} \|_2$

$j := j + 1$
endwhile

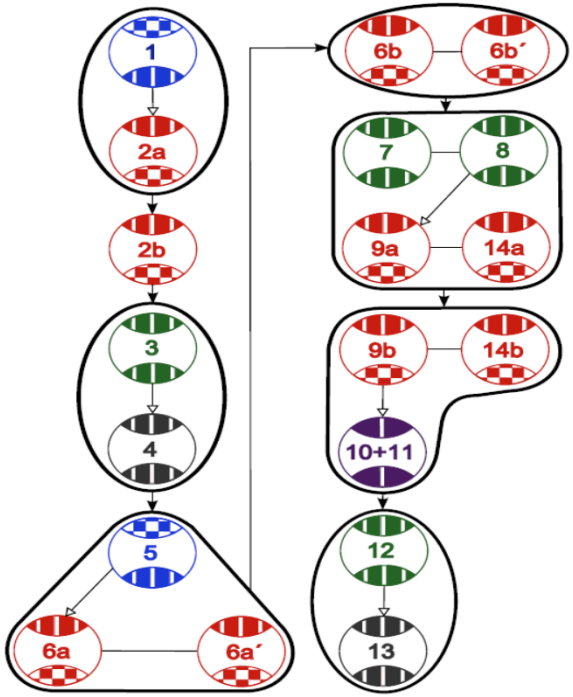


Case Studies

CG:



BiCGStab:



Experiments

UFMC and Laplace problem

Matrix	n_z	n	n_z/n
BMWCRA1_1	10,641,602	148,770	71.53
CRANKSEG_2	14,148,858	63,838	221.63
F1	26,837,113	343,791	78.06
INLINE_1	38,816,170	503,712	77.06
LDOOR	42,493,817	952,203	44.62
AUDIKW_1	77,651,847	943,645	82.28
FEM_3DTH2	3,489,300	147,900	23.59

Matrix	n_z	n	n_z/n
A100	6,940,000	1,000,000	6.94
A126	13,907,370	2,000,376	6.94
A159	27,986,067	4,019,679	6.94
A200	55,760,000	8,000,000	6.94
A252	111,640,032	16,003,001	6.94

Server:

- Intel core i7-3770K (4 cores, 3.5 GHz) with 16 GB DDR3 RAM
- NVIDIA Kepler K20c GPU (2,496 CUDA cores, 706 MHz) with 5 GB GDDR5 RAM
- National Instruments DAS (NI9205 module and the NIcDAQ-9178)

Experiments

Software

- Single-precision arithmetic
- Solvers: CG, BiCG, BiCGStab with/out preconditioning
- SpMV: scalar CSR, vector CSR, ELL
- Implementations of solvers:
  - CUBLASL: Plain implementation + ad-hoc SpMV + CUBLAS
  - CUBLASN: Maintain scalars in GPU (new CUBLAS interface)
  - CUDA: Replace CUBLAS by ad-hoc kernels (including two-stage DOT)
  - MERGE: Applies fusion
  - MERGE_10: Check convergece every 10 iterations
- Execution modes: CUDA polling/blocking

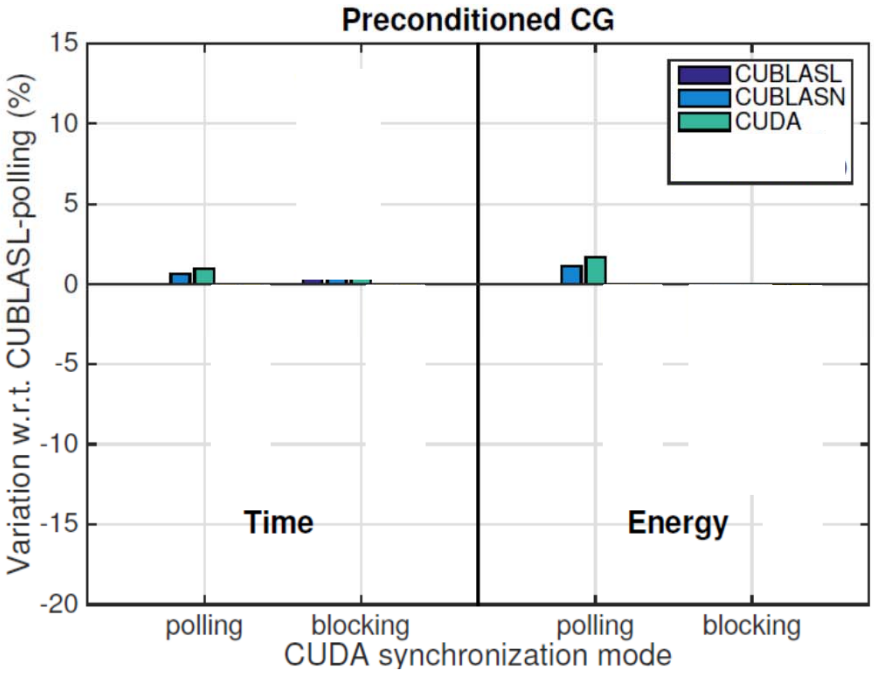
Experiments

3 solvers *x* with/out preconditioning *x* 3 SpMV *x* 5 implementations *x* 2 exec. modes *x* 12 matrices…

- Variations of time/energy w.r.t. CUBLASL in polling mode
- Average results for all matrices giving the same weight to all cases
- Vector CSR for UFMC and ELL for Laplace (best options)

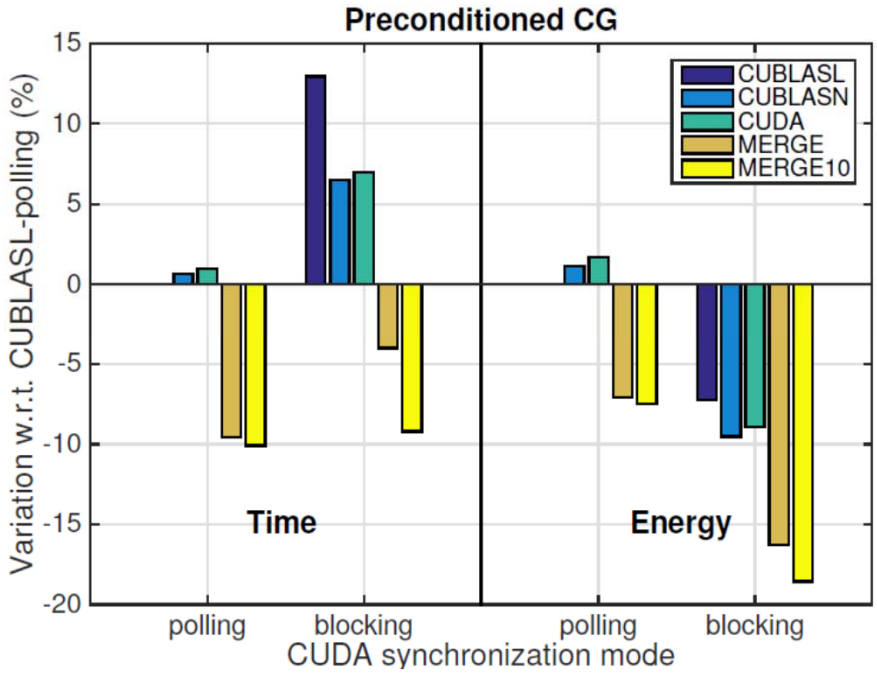
Experiments

- In polling mode, CUBLASL and CUDA incur small overhead in time and energy
- Blocking mode trades off time for energy



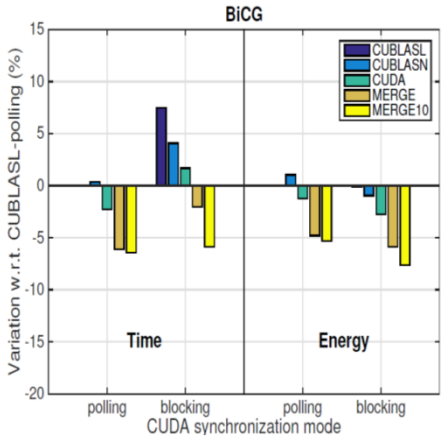
Experiments

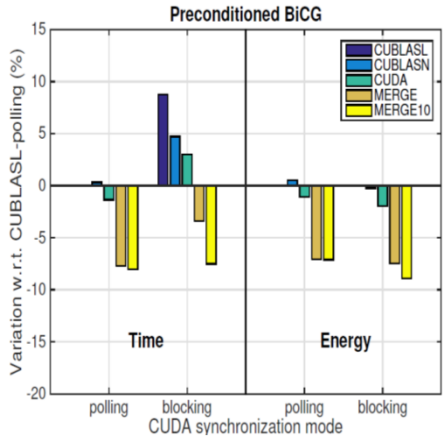
- MERGE and MERGE 10 reduce both time and energy
- Polling mode prioritizes reduction of time vs energy
- Blocking mode prioritizes energy vs time

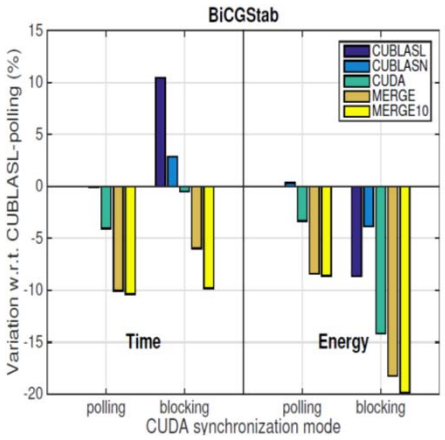


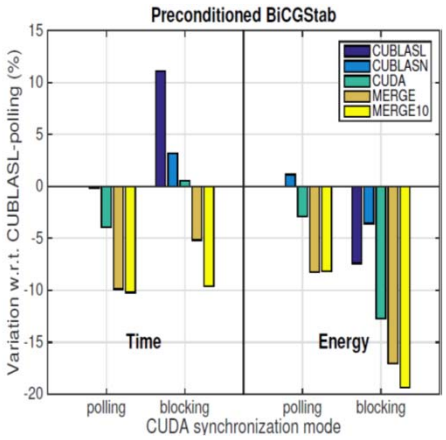
Experiments

Similar trends for BiCG, BiCGStab









Concluding Remarks

Systematic fusion to derive efficient versions of three popular iterative solvers (with/out preconditioning) for sparse linear systems

- Analysis of type of thread accesses on kernel input/outputs
- Identification of data dependencies
- Eager fusion
- Divide DOT into two stages
- Reduce #kernels to improve performance/energy consumption
  - 10 → 5 for PCG, 13 → 5 for PBiCG and 14 → 8 for PBiCGStab
  - Remarkable energy savings when executed in blocking mode
  - Match performance of original versions when executed in polling mode

THANKS… and QUESTIONS?