Harnessing CUDA Dynamic Parallelism for the Solution of Sparse Linear Systems

José ALIAGA , [a,1] Davor DAVIDOVIĆ [b], Joaquín PÉREZ [a], and
Enrique S. QUINTANA-ORTÍ [a],

[a] *Dpto. Ingeniería Ciencia de Computadores, Universidad Jaume I, Castellón (Spain)*
[b] *Institut Ruđer Bošković, Centar za Informatiku i Računarstvo - CIR, Zagreb (Croatia)*

Abstract. We leverage CUDA dynamic parallelism to reduce execution time while significantly reducing energy consumption of the Conjugate Gradient (CG) method for the iterative solution of sparse linear systems on graphics processing units (GPUs). Our new implementation of this solver is launched from the CPU in the form of a single "parent" CUDA kernel, which invokes other "child" CUDA kernels. The CPU can then continue with other work while the execution of the solver proceeds asynchronously on the GPU, or block until the execution is completed. Our experiments on a server equipped with an Intel Core i7-3770K CPU and an NVIDIA "Kepler" K20c GPU illustrate the benefits of the new CG solver.

Keywords. Graphics processing units (GPUs), CUDA dynamic parallelism, sparse linear systems, iterative solvers, high performance, energy efficiency

Introduction

The discretization of partial differential equations (PDEs) often leads to large-scale linear systems of the form $Ax = b$, where the coefficient matrix $A \in \mathbb{R}^{n\times n}$ is *sparse*, $b \in \mathbb{R}^n$ contains the independent terms, and $x \in \mathbb{R}^n$ is the sought-after solution. For many problems (especially those associated with 3-D models), the size and complexity of these systems have turned iterative projection methods, based on Krylov subspaces, into a highly competitive approach compared with direct methods [1].

The Conjugate Gradient (CG) method is one of the most efficient Krylov subspace-based algorithms for the solution of sparse linear systems when the coefficient matrix is symmetric positive definite (s.p.d.) [1]. Furthermore, the structure and numerical kernels arising in this iterative solver are representative of a wide variety of efficient solvers for other specialized types of sparse linear systems.

When the target platform is a heterogeneous server consisting of a multicore processor plus a graphics processing unit (GPU), a conventional implementation of the CG method completely relies on the GPU for the computations, and leaves the general-purpose multicore processor (CPU) in charge of controlling the GPU only. The reason is that this type of iterative solvers is composed of fine-grain kernels, which exhibit a low ratio between computation and data accesses (in general, O(1)). In this scenario, com-

[1]Corresponding Author: Dpto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón (Spain); E-mail: aliaga@icc.uji.es.

municating data via a slow PCI-e bus mostly blurs the benefits of a CPU-GPU collaboration. In addition, in [2] we demonstrated the negative effect of CPU-GPU synchronization when the body of the iterative loop in the CG solver is implemented via calls to the GPU kernels, e.g. in CUBLAS/cuSPARSE. The cause is that, in such implementation, the CPU thread in control of the GPU repeatedly invokes fine-grain CUDA kernels of low cost and short duration, resulting in continuous stream of kernel calls that prevents the CPU from entering an energy-efficient C-state.

Our solution to alleviate these performance and energy overheads in [2] was to *fuse* (i.e. *merge*) CUDA kernels in order to decrease their number, thus reducing the volume of CPU-GPU synchronizations. The main contribution of this paper lies in the investigation of *dynamic parallelism* (DP) [3], as a complementary/alternative technique to achieve the same effect with a more reduced programming effort. Concretely, this work provides a practical demonstration of the benefits of DP on a solver like the CG method, representative of many other sparse linear system solvers as well as, in general, fine-grain computations. Our experimental evaluation of this algorithm on a platform equipped with an Intel Xeon processor and an NVIDIA "Kepler" GPU reports savings in both execution time and energy consumption, respectively of 3.65% and 14.23% on average, for a collection of problems.

DP is a recent technology introduced recently in the CUDA programming model, and is available for NVIDIA devices with compute capability 3.5 or higher. With DP, a child CUDA kernel can be called from within a parent CUDA kernel and then optionally synchronized on the completion of that child CUDA kernel. Some research on DP pursue the implementation of clustering and graph algorithms on GPUs [4,5,6], and a more complete analysis of unstructured applications on GPUs appears in [7]. This technology is also included as a compiler technique [8] to handle nested parallelism in GPU applications. DP is also used to avoid deadlocks in intra-GPU synchronization, reducing the energy consumption of the system [9].

The rest of the paper is structured as follows. In section 1 we briefly review the CG method and the fusion-based approach proposed in our earlier work. In section 2 we present the changes required to a standard implementation of the CG solver in order to efficiently exploit DP. In section 3 we evaluate the dynamic(-parallel) implementation of the new solver, and in section 4 we summarize the insights gained from our study.

1. Fusions in the CG method

1.1. Overview

Figure 1 offers an algorithmic description of the CG solver. Concerning the computational effort of the method, in practice the cost of the iteration loop is dominated by the sparse matrix-vector multiplication (SPMV) involving A. In particular, given a sparse matrix A with n_z nonzero entries, the cost of the SPMV in O1 is roughly $2n_z$ floating-point arithmetic operations (flops), while the vector operations in the loop body (O2, O3, O4, O5 and O8) require $O(n)$ flops each.

The dependencies between the operations in the body of the iterative loop of the CG method dictate a partial order for their execution. Specifically, at the $(j+1)$-th iteration,

```
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 product
    $\beta_j := \zeta_j / \sigma_j$            O6. Scalar op
    $\sigma_{j+1} := \zeta_j$                  O7. Scalar op
    $p_{j+1} := z_j + \beta_j p_j$             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
```

Figure 1. Algorithmic formulation of the CG method. In general, we use Greek letters for scalars, lowercase for vectors and uppercase for matrices. Here, $\tau_{\max}$ is an upper bound on the relative residual for the computed approximation to the solution.

$$\ldots \to \mathrm{O8} \to \overbrace{\mathrm{O1} \to \mathrm{O2} \to \mathrm{O4} \to \mathrm{O5} \to \mathrm{O6} \to \mathrm{O7} \to \mathrm{O9}}^{(j+1)\text{-th iteration}} \to \mathrm{O1} \to \ldots$$

must be computed in that order, but O3 and O8 can be computed any time once O2 and O6 are respectively available.

1.2. Merging kernels in CG

In [2], we exploited that two CUDA kernels related by a RAW (read-after-write) dependency [10], dictated by a vector v that is an output of/input to the first/second kernel, can be merged if *(i)* both kernels apply the same mapping of threads to the elements of v shared (exchanged) via registers; *(ii)* both kernels apply the same mapping of thread blocks to the vector elements shared (exchanged) via shared memory; and *(iii)* a global barrier is not necessary between the two kernels.

In addition, in [2] we developed a tailored implementation of the kernel DOT that consisted of two stages, say DOT$^{\mathrm{M}}_{\mathrm{ini}}$ and DOT$^{\mathrm{M}}_{\mathrm{fin}}$, so that the first stage can be efficiently merged with a prior dependent kernel. In particular, the first stage was implemented as a GPU kernel which performs the costly element-wise products and subsequent reduction within a thread block, producing a partial result in the form of a temporary vector with one entry per block. This was followed by a *routine* DOT$^{\mathrm{M}}_{\mathrm{fin}}$ which completed the operation by repeatedly reducing the contents of this vector into a single scalar via a sequence of calls to GPU kernels; see [2] for details.

Figure 2 illustrates the fusions that were derived in our previous work for the CG solver when SpMV is based on the CSR scalar or ELL formats [11]. The node colors distinguish between three different operation types: SpMV, DOT and AXPY-like (AXPY/XPAY). As argued earlier, each DOT operation (O2 and O5) is divided into two stages (a or b, corresponding respectively to kernel DOT$^{\mathrm{M}}_{\mathrm{ini}}$ and routine DOT$^{\mathrm{M}}_{\mathrm{fin}}$) in order to facilitate the fusion of the first one with a previous kernel. The fusions are encircled by thick lines and designate three macro-kernels: {O1-O2a}, {O3-O4-O5a}, {O5b-O6-O7-O9}; plus two single-node (macro-)kernels: {O2b} and {O8}. The arrowless lines connect independent kernels (e.g., O3 and O4) and the arrows identify dependencies inside macro-kernels (e.g., from O1 to O2a) and between them (e.g., from {O1-O2a}

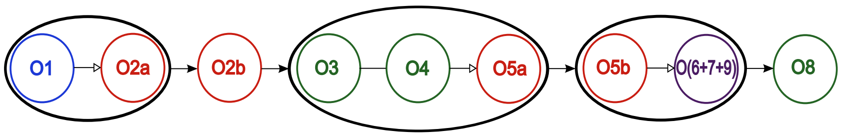

Figure 2. Fusions for the CG solver with SpMV based on the scalar CSR or ELL format.

to {O2b}). When SpMV employs the CSR vector format [11] the fusion graph differs from that in Figure 2 in that O1 and O2a cannot be merged.

This specialized formulation of the CG solver merged multiple numerical operations, reducing the number of synchronizations and data transfers, in turn yielding a more efficient hardware utilization. The experimental evaluation with a varied benchmark of linear systems from the University of Florida Matrix Collection and the 3D Laplace problem, using a server equipped with an Intel Core i7-3770K processor and an NVIDIA GeForce GTX480 GPU, revealed remarkable CPU energy savings and minor improvements on runtime, with respect to a plain implementation of the CG method based on the use of CUBLAS kernels and the CUDA polling synchronization mode.

2. Exploiting DP to Enhance CG

In principle, kernel fusion and DP are orthogonal techniques that can be applied independently or in combination. As described next, our *dynamic version* of CG integrates specialized implementations of DOT, AXPY and XPAY, which are more efficient than their "fusible" counterparts previously developed for the *merge version*.

2.1. Two-stage dynamic DOT

Our previous implementation of DOT divided this operation into two stages, $\text{DOT}^{\text{M}}_{\text{ini}}$ and $\text{DOT}^{\text{M}}_{\text{fin}}$, with the former one implemented as a CUDA kernel and the second being a routine that consists of a loop which invoked a CUDA kernel per iteration. The problem with this approach is that, if integrated with DP, $\text{DOT}^{\text{M}}_{\text{fin}}$ involves a sequence of nested calls to CUDA kernels from inside the GPU and, in practice, incurs a large overhead. In order to avoid this negative effect, we redesigned DOT as a two-stage procedure, $\text{DOT}^{\text{D}}_{\text{ini}}$ and $\text{DOT}^{\text{D}}_{\text{fin}}$, but with each stage implemented as a single CUDA kernel. One main difference between $\text{DOT}^{\text{D}}_{\text{ini}}$ and $\text{DOT}^{\text{M}}_{\text{ini}}$ is that the former cannot be merged with other kernels. However, we note that the "dynamic" version of the first stage is intended to be used in combination with DP and, therefore, reducing the number of kernels is no longer a strong urge (though it may still be convenient).

Figures 3 and 4 illustrate the implementation of the GPU kernels $\text{DOT}^{\text{D}}_{\text{ini}}$ and $\text{DOT}^{\text{D}}_{\text{fin}}$, respectively. There, `n` specifies the length of the vectors, `x` and `y` are the vectors involved in the reduction. The first kernel is invoked as

```
DOT_D_ini <<NumBlk, BlkSize, sizeof(float)*BlkSize>> (n, x, y, valpha);
```

and reduces the two vectors into `NumBlk`=256 partial results, stored upon completion in `valpha`. We note that, for performance reasons, this kernel spawns `GrdSize` = `NumBlk` · `BlkSize`= $256 \cdot 192$ threads, and each thread, processes two entries (one at `threadId` and a second at `threadId+BlkSize`) per iteration and per chunk of 2·`GrdSize` elements

```
__global__ void DOT_D_ini(int n, float *x, float *y, float *valpha) {
  extern __shared__ float vtmp[];

  // Each thread loads two elements from each chunk
  // from global to shared memory
  unsigned int   tid       = threadIdx.x;
  unsigned int   NumBlk    = gridDim.x;     // = 256
  unsigned int   BlkSize   = blockDim.x;    // = 192
  unsigned int   Chunk     = 2 * NumBlk * BlkSize;
  unsigned int   i         = blockIdx.x * (2 * BlkSize) + tid;
  volatile float *vtmp2    = vtmp;

  // Reduce from n to NumBlk * BlkSize elements. Each thread
  // operates with two elements of each chunk
  vtmp[tid] = 0;
  while (i < n) {
    vtmp[tid] += x[i] * y[i];
    vtmp[tid] += (i+BlkSize < n) ? (x[i+BlkSize] * y[i+BlkSize]): 0;
    i         += Chunk;
  }
  __syncthreads();

  // Reduce from BlkSize=192 elements to 96, 48, 24, 12, 6, 3 and 1
  if (tid < 96) { vtmp[tid] += vtmp[tid + 96]; } __syncthreads();
  if (tid < 48) { vtmp[tid] += vtmp[tid + 48]; } __syncthreads();
  if (tid < 24) {
    vtmp2[tid] += vtmp2[tid + 24]; vtmp2[tid] += vtmp2[tid + 12];
    vtmp2[tid] += vtmp2[tid + 6 ]; vtmp2[tid] += vtmp2[tid + 3 ];
  }

  // Write result for this block to global mem
  if (tid == 0) valpha[blockIdx.x] = vtmp[0] + vtmp[1] + vtmp[2];
}
```

Figure 3. Implementation of kernel DOT$^{\mathrm{D}}_{\mathrm{ini}}$.

of the vectors, yielding a coalesced access to their entries; see Figure 5. The subsequent invocation to kernel

```
DOT_D_fin <<NumBlk2, BlkSize2, sizeof(float)*BlkSize2>> (valpha);
```

then produces the sought-after scalar result into the first component of this vector. The grid and block dimensions `NumBlk` = 256, `BlkSize` = 192, `NumBlk2` = 1, `BlkSize2` = `NumBlk` = 256 passed for the kernel launches were experimentally determined, except for `BlkSize` which was set to the number of CUDA cores per SMX (streaming multiprocessor).

2.2. *Two-stage dynamic* AXPY/XPAY

For performance reasons, the AXPY and XPAY have been also reorganized in the dynamic version of CG so that each CUDA thread operates with a pair of elements of each vectors. Figure 6 offers the code for the former, with the second operation being implemented in an analogous manner. There, `n` specifies the length of the vectors, `x` and `y` are the vectors involved in the operation, and `alpha` is the scalar. The call to this kernel is done as

```
__global__ void DOT_D_fin(float *valpha) {
  extern __shared__ float vtmp[];

  // Each thread loads one element from global to shared mem
  unsigned int  tid = threadIdx.x;
  volatile float *vtmp2 = vtmp;

  vtmp[tid] = valpha[tid]; __syncthreads();

  // Reduce from 256 elements to 128, 64, 32, 16, 8, 2 and 1
  if (tid < 128) { vtmp[tid] += vtmp[tid + 128]; } __syncthreads();
  if (tid < 64)  { vtmp[tid] += vtmp[tid + 64 ]; } __syncthreads();
  if (tid < 32)  {
    vtmp2[tid] += vtmp2[tid + 32]; vtmp2[tid] += vtmp2[tid + 16];
    vtmp2[tid] += vtmp2[tid + 8 ]; vtmp2[tid] += vtmp2[tid + 4 ];
    vtmp2[tid] += vtmp2[tid + 2 ]; vtmp2[tid] += vtmp2[tid + 1 ];
  }

  // Write result for this block to global mem
  if (tid == 0) valpha[blockIdx.x] = *vtmp;
}
```

Figure 4. Implementation of kernel DOT$^{\mathrm{D}}_{\mathrm{fin}}$.

```
AXPY_D <<NumBlk3, BlkSize3/2>> (n, alpha, x, y);
```

with NumBlk3=$\lceil$n/BlkSize3$\rceil$ and BlkSize3 = 256. The same values are also used for the dynamic implementation of XPAY.

Finally, our dynamic CG solver merges O3 and O4 into a single macro-kernel. creating another macro-kernel with the scalar operations (O6, O7 and O9) and the second stage of the last DOT (O5b)

3. Experimental Evaluation

We next expose the performance and energy of the dynamic CG solver compared with our previous implementations [2]. For this purpose, we employ several sparse matrices from the University of Florida Matrix Collection (UFMC)[2] and a difference discretization of the 3D Laplace problem; see Table 1. For all cases, the solution vector was chosen to have all entries equal 1, and the independent vector was set to $b = Ax$. The iterative solvers were initialized with the starting guess $x_0 = 0$. All experiments were done using IEEE single precision arithmetic. While the use of double precision arithmetic is in general mandatory for the solution of sparse linear systems, the use of mixed single/double-precision in combination with iterative refinement leads to improved execution time and energy consumption when the target platform is a GPU accelerator.

The target architecture is a Linux server (CentOS release 6.2 with kernel 2.6.32 with CUDA v5.5.0) equipped with a single Intel Core i7-3770K CPU (3.5 GHz, four cores) and 16 Gbytes of DDR3 RAM, connected via a PCI-e 2.0 bus to an NVIDIA "Kepler"

[2]http://www.cise.ufl.edu/research/sparse/matrices/

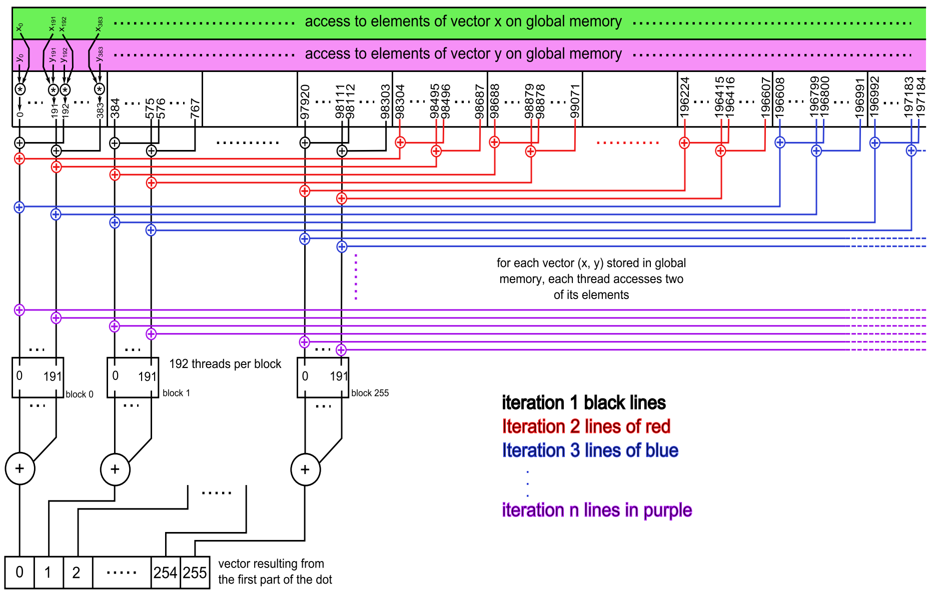
access to elements of vector x on global memory
access to elements of vector y on global memory
0
191
192
383
384
575
576
767
97920
98111
98112
98303
98304
98495
98496
98687
98688
98879
98878
99071
196224
196415
196416
196607
196608
196799
196800
196991
196992
197183
197184
for each vector (x, y) stored in global memory, each thread accesses two of its elements
192 threads per block
block 0
block 1
block 255
iteration 1 black lines
Iteration 2 lines of red
Iteration 3 lines of blue
iteration n lines in purple
vector resulting from the first part of the dot
0
1
2
254
255

Figure 5. Implementation of $\text{DOT}^{\text{D}}_{\text{ini}}$.

```
__global__ void AXPY_D(int n, float *alpha, float *x, float *y) {
  unsigned int NumBlk   = gridDim.x;
  unsigned int BlkSize  = blockDim.x;
  unsigned int i        = blockIdx.x * (2 * BlkSize) + threadIdx.x;
  unsigned int Chunk    = 2 * NumBlk * BlkSize;

  while (i < n) {
    y[i] +=  *alfa * x[i];
    if (i + BlkSize < n) y[i + BlkSize] += *alfa * x[i + BlkSize];
    i += Chunk;
  }
}
```

Figure 6. Implementation of kernel AXPY^{D}.

K20c GPU (compute capability 3.5, 706 MHz, 2,496 CUDA cores) with 5 GB of DDR5 RAM integrated into the accelerator board. Power was collected using a *National Instruments* data acquisition system, composed of the NI9205 module and the NIcDAQ-9178 chassis, and plugged to the lines that connect the power supply unit with motherboard and GPU.

Our experimental evaluation included four implementations of the CG solver:

- CUBLASL is a plain version that relies on CUBLAS kernels from the legacy programming interface of this library, combined with *ad-hoc* implementations of SPMV. In this version, one or more scalars may be transferred between the main memory and the GPU memory address space each time a kernel is invoked and/or

Matrix	Acronym	n_z	n	n_z/n
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

Table 1. Description and properties of the test matrices from the UFMC and the 3D Laplace problem.

CUDA mode	Implementation	Time			Energy		
		Min	Max	Avg.	Min	Max	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	-0.89	-3.07	-1.71	-1.42	5.03	0.62
	DYNAMIC	-1.54	-4.76	-3.65	-1.17	-3.32	-2.58
Blocking	CUBLASL	0.62	12.88	7.15	-3.30	-13.48	-10.85
	CUDA	0.78	9.39	4.74	-4.45	-12.62	-10.70
	MERGE	-0.59	-1.70	-1.06	-8.31	-13.96	-12.47
	DYNAMIC	-1.54	-4.71	-3.65	-13.73	-14.50	-14.23

Table 2. Minimum, maximum and average variations (in %) of execution time and energy consumption for CG with respect to the baseline.

its execution is completed.

- CUDA replaces the CUBLAS (vector) kernels in the previous version by our *ad-hoc* implementations.
- MERGE applies the fusions described in Section 1, including the two-stage $\text{DOT}^{\text{M}}_{\text{ini}}$ $+\text{DOT}^{\text{M}}_{\text{fin}}$.
- DYNAMIC exploits DP including the two-stage dynamic implementations of DOT, AXPY and XPAY introduced in Section 2 and merging O3 and O4 into a single macro-kernel.

Furthermore, we execute these configurations under the CUDA polling and blocking synchronization modes. We evaluated three different implementations of SPMV, scalar CSR, vector CSR and ELL [11], but only report results for the second one (vector CSR) which was experimentally determined to be the best option for most of the matrix cases.

Figure 7 reports the time and energy variations of the CUDA, MERGE and DYNAMIC versions of the CG solver with respect to CUBLASL executed in the CUDA polling synchronization mode (*baseline case*) for each matrix case. Table 2 summarizes these results via minimum, maximum and average numbers (with the former two corresponding to the largest and smallest in absolute value).

Let us analyze first the CUBLASL implementation executed in blocking mode. Compared with the same routine executed in polling mode (i.e, the baseline case), we observe an appealing reduction of the energy consumption, -10.85% on average, though it comes at the cost of a noticeable increase in the execution time, around 7.15% on average. The CUDA version of the solver executed in polling mode incurs a very small overhead in execution time with respect to the baseline (0.21% on average) and a slightly larger one in

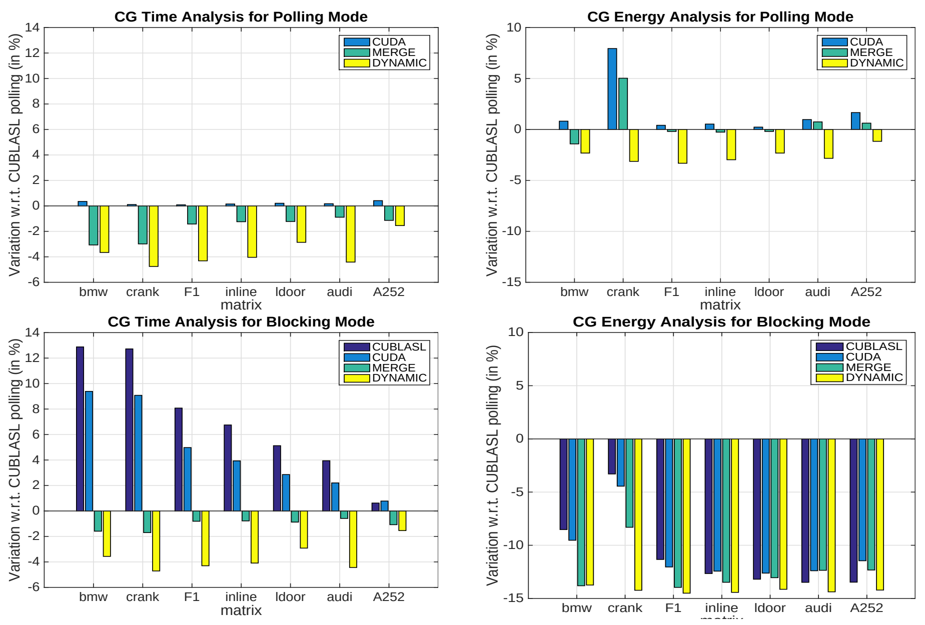

Figure 7. Variations (in %) of execution time and energy consumption of the CG solver (left and right, respectively) with respect to the baseline.

energy (1.79% on average). Moreover, when executed in blocking mode, the CUBLASL and CUDA implementations offer a close behavior, with a considerable increase in execution time that neutralizes the benefits of the notable reduction in energy. The MERGE variant of the solver combines the speed of a polling execution with the energy efficiency of a blocking one. In particular, this variant executed in blocking mode slightly reduces the average execution time (by -1.06%) while extracting much of the energy advantages of this mode (reduction of -12.47% on average). Finally, the DYNAMIC implementation outperforms all other variants, including MERGE, obtaining the largest reduction in both time and energy (respectively, -3.65% and -14.23% on average). In addition, the DYNAMIC version of the solver does not require the complex reorganization of the code entailed by the MERGE case.

4. Concluding Remarks

We have presented a CUDA implementation of the CG method for the solution of sparse linear systems that exploits DP as means to produce a more energy efficient solution. With this new implementation, the CPU invokes a single "parent" CUDA kernel in order to launch the CG solver on the GPU, and can then proceed asynchronously to perform other work or be simply put to sleep via the CUDA blocking synchronization mode and the CPU C-states. The GPU is then in charge of executing the complete solver, with the parent kernel calling other "child" CUDA kernels to perform specific parts of the job. In order to improve efficiency, we have redesigned the implementation of the key vector operations arising in CG, concretely the dot product and axpy-like operations, into two-stage CUDA kernels. This is particular important for the former operation in order to avoid that the exploitation of DP results in a hierarchy of "nested" invocations to other CUDA kernels (i.e., a multilevel structure of parents and children).

The experimentation on a platform with a recent Intel Core i7-3770K CPU and an NVIDIA "Kepler" K20c GPU reports the superiority of the dynamic CG solver, which outperforms our previous fusion-based implementation, in both execution time and energy consumption. From the programming point of view, the dynamic version also presents the important advantage of being more modular, as it does not require the major reorganization of CG, via kernel fusions, that were entailed by the fusion-based implementation.

We have applied similar techniques to iterative solvers based on BiCG and BiCGStab, as well as variants of these and CG that include a simple, Jacobi-based preconditioner, with similar benefits on performance and energy efficiency. The gains that can be obtained with DP heavily depend on the granularity of the CUDA kernels and, as the complexity of the preconditioner grows, we can expect that the positive impact of DP decreases.

Acknowledgments

This research has been supported by EU under projects EU FP7 318793 (Exa2Green), EU FEDER, the COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS), and the project TIN2011-23283 of the *Ministerio de Economía y Competitividad.*

References

[1] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* SIAM, 2nd edition, 2003.

[2] J. I. Aliaga, J. Pérez, E. S. Quintana-Ortí, and H. Anzt. Reformulated Conjugate Gradient for the Energy-Aware Solution of Linear Systems on GPUs. In *42nd Int. Conference on Parallel Processing (ICPP)*, pages 320–329, 2013.

[3] NVIDIA Coorporation. Dynamic parallelism in CUDA. `http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief\_Dynamic\_Parallelism\_in\_CUDA.pdf`, February 2015.

[4] Jeffrey DiMarco and Michela Taufer. Performance impact of dynamic parallelism on different clustering algorithms. volume 8752, pages 87520E–87520E–8, 2013.

[5] Fei Wang, Jianqiang Dong, and Bo Yuan. Graph-based substructure pattern mining using cuda dynamic parallelism. In *IDEAL-2013*, volume 8206 of *Lecture Notes in Computer Science*, pages 342–349. Springer Berlin Heidelberg, 2013.

[6] Jianqiang Dong, Fei Wang, and Bo Yuan. Accelerating birch for clustering large scale streaming data using cuda dynamic parallelism. In *IDEAL-2013*, volume 8206 of *Lecture Notes in Computer Science*, pages 409–416. Springer Berlin Heidelberg, 2013.

[7] Jin Wang and Sudhakar Yalamanchili. Characterization and analysis of dynamic parallelism in unstructured gpu applications. In *2014 IEEE International Symposium on Workload Characterization*, 2014.

[8] Yi Yang, Chao Li, and Huiyang Zhou. Cuda-np: Realizing nested thread-level parallelism in gpgpu applications. *Journal of Computer Science and Technology*, 30(1):3–19, 2015.

[9] L. Oden, B. Klenk, and H. Froning. Energy-efficient stencil computations on distributed gpus using dynamic parallelism and gpu-controlled communication. In *Energy Efficient Supercomputing Workshop (E2SC), 2014*, pages 31–40, 2014.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Pub., San Francisco, 2003.

[11] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corp., December 2008.