Enhancing the Performance of Conjugate Gradient Solvers on Graphic Processing Units

Maryam Mehri Dehnavi, David M. Fernández, and Dennis Giannacopoulos

Electrical and Computer Engineering Department, McGill University Montreal, QC H3A 2A7 Canada

A study of the fundamental obstacles to accelerate the preconditioned conjugate gradient (PCG) method on modern graphic processing units (GPUs) is presented and several techniques are proposed to enhance its performance over previous work independent of the GPU generation and the matrix sparsity pattern. The proposed enhancements increase the performance of PCG up to 23 times compared to vector optimized PCG results on modern CPUs and up to 3.4 times compared to previous GPU results.

Index Terms—**Computer architecture, conjugate gradients (CGs), graphic processing units (GPUs), parallel processing.**

I. Introduction

REAL world electromagnetic problems constantly demand more precise and sophisticated simulations in reasonable time frames. To meet such demands in modern finite element method (FEM) applications, programmers must efficiently exploit new technological advancements in modern computing systems. Graphic processing units (GPUs) have evolved very quickly over the last few years and significantly overwhelm CPU specifications in both raw power and memory bandwidth [1]. To benefit from the pervasive computing resources in a GPU, compute intensive data-parallel sections of large problems should be optimized to run on the GPU architecture.

This paper focuses on enhancing the performance of the preconditioned conjugate gradient (PCG) algorithm [4], a popular sparse linear solver in FEM using current GPU processors. Efficient techniques to parallelize PCG on GPUs are presented that overcome the main limitations imposed by both the PCG algorithm (namely poor data locality and sequential execution), and the programming constraints of modern GPUs (e.g., efficient use of different GPU resources, minimizing data communication, hiding memory access latencies and reducing the number of kernel calls). The effectiveness of these techniques is demonstrated using a range of matrices and speedup results are compared with other state-of-the-art PCG multicore and GPU implementations.

II. GPU Architecture

Initially driven by the demand for powerful high-definition 3-D graphics, modern GPUs have become massively parallel, multithreaded architectures. Easy to learn APIs (Application Programming Interfaces) such as compute unified device architecture (CUDA [3]) has enabled the acceleration of modern scientific applications via massive multithreading. In particular, NVIDIA GPUs offer important computing power for these applications. Fig. 1 shows the general architecture of NVIDIA graphic cards. Scalar processors (SPs) are the basic processing units of the architecture and are clustered in groups of eight called streaming multiprocessors (SMs).

Manuscript received May 31, 2010; accepted September 18, 2010. Date of current version April 22, 2011. Corresponding author: M. Mehri (e-mail: maryam.mehridehnavi@mail.mcgill.ca).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TMAG.2010.2081662

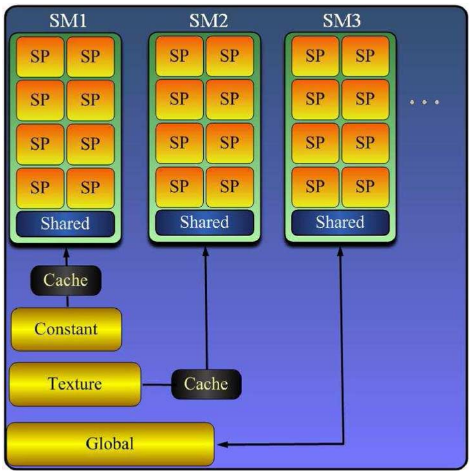

Fig. 1. NVIDIA GPU architecture.

Sections of an application that exhibit rich data parallelism are scheduled to run on the GPU. Executing a parallel section on the GPU using CUDA involves: a) transferring required data to GPU global memory; b) launching the device (GPU) kernel; and c) transferring results back to host memory. Threads inside a kernel are grouped into thread blocks, which are executed on SMs. Threads in a block communicate via fast shared memory, but threads in different blocks communicate through long latency global memory. Major challenges in optimizing an application on GPUs are: global memory access latency, different execution paths in each warp (32 consecutive threads in a block) namely *thread divergence*, communication and synchronizations between threads in different blocks and resource utilization.

III. Preconditioned Conjugate Gradient (PCG)

The conjugate gradient (CG) algorithm is one of the most popular iterative linear solvers available today, mainly due to its fast convergence, constant decrease in error-per-iteration and efficient memory usage [4]. Before introducing the parallelization and performance enhancing techniques, one must choose an appropriate PCG version with good parallelization properties as presented in the next section.

A. Choosing a PCG Algorithm

Many variations of the PCG algorithm exist, depending on their formulation. In this paper, we implemented a classical PCG

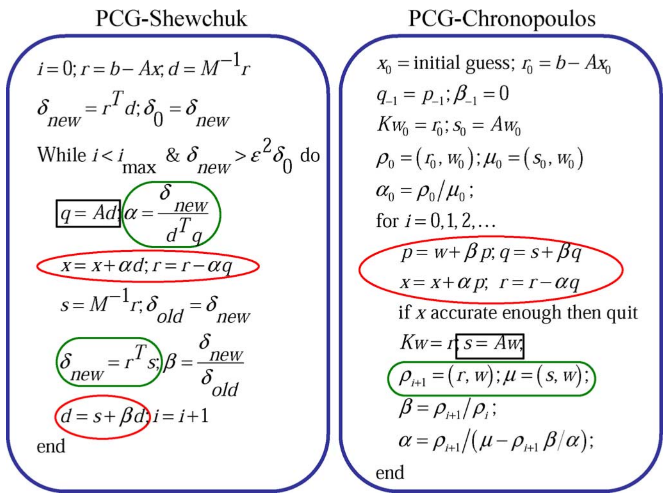
PCG-Shewchuk
$i=0; r=b-Ax; d=M^{-1}r$
$\delta_{new}=r^T d; \delta_0=\delta_{new}$
While $i<i_{max}$ & $\delta_{new}>\varepsilon^2\delta_0$ do
$\boxed{q=Ad}$ $\alpha=\frac{\delta_{new}}{d^T q}$
$x=x+\alpha d; r=r-\alpha q$
$s=M^{-1}r; \delta_{old}=\delta_{new}$
$\delta_{new}=r^T s; \beta=\frac{\delta_{new}}{\delta_{old}}$
$d=s+\beta d; i=i+1$
end
PCG-Chronopoulos
$x_0=$ initial guess; $r_0=b-Ax_0$
$q_{-1}=p_{-1}; \beta_{-1}=0$
$Kw_0=r_0; s_0=Aw_0$
$\rho_0=(r_0,w_0); \mu_0=(s_0,w_0)$
$\alpha_0=\rho_0/\mu_0$;
for $i=0,1,2,\ldots$
$p=w+\beta p; q=s+\beta q$
$x=x+\alpha p; \ r=r-\alpha q$
if x accurate enough then quit
$Kw=r; \boxed{s=Aw}$
$\rho_{i+1}=(r,w); \mu=(s,w)$;
$\beta=\rho_{i+1}/\rho_i$;
$\alpha=\rho_{i+1}/(\mu-\rho_{i+1}\beta/\alpha)$;
end

Fig. 2. Highlighting several bottleneck operations in PCG Shewchuk [4] versus PCG Chronopoulos [2].

algorithm [4] and a variation presented in [2] with better data locality that minimizes the number of kernel calls, the GPU global memory loads, and the communication overhead. Fig. 2 presents both algorithms highlighting sections in the main iteration loop where vectors are loaded for the sparse matrix vector multiplication (SMVM), SAXPY (vector updates, $y \leftarrow ax+y$) and dot product operations.

The main advantages of the Chronopoulos variant of the PCG algorithm compared to the Shewchuk method are as follows:

- In the PCG-Chronopoulos version vectors are loaded in the same place within the main loop as opposed to across the whole loop for the Shewchuk version. This property allows multiple operations to reuse data while on shared memory, reducing long latency memory accesses and exhibiting better data locality.
- Dot products are clustered together in the Chronopoulos variant reducing the number of synchronization steps on both the GPU and the CPU.
- Efficient partitioning of vector and matrix values enables coalesced loading of data and maximum GPU resource utilization during PCG kernel calculations.

B. *Previous Work*

Accelerating the PCG algorithm on massively parallel hardware platforms, especially GPUs, is very challenging due to the sequential nature of the algorithm. Buatois *et al.* [1] accelerated the CG solver on GPUs using the blocked compressed sparse row storage (BCSR) format. Their algorithm is optimized for a limited set of matrices with specific sparsity patterns. Wiggers *et al.* [5] reorder matrix rows to decrease the execution time of the SMVM kernel for CG. Sorting rows increases preprocessing and execution time on the CPU. In [6] and [7] a mixed precision iterative refinement algorithm is proposed for the CG. The single precision inner solver in their method is the most time consuming kernel in the overall solution and accelerating its execution is the major focus of our work.

The performance of SMVM using various compressed storage formats on GPUs has been studied in [8]. Using a decision based method, [9] chooses the best performing storage format for SMVM from [8] prior to executing the CG algorithm, at the expense of storing (generating) several copies of the matrices in the various storage formats. Formats such as JDS [9], HYB [8], ELL [8], BCSR [1] require extra preprocessing to benefit from the GPU processors (sorting rows, blocking nonzero values, redundant padding, etc.) that are not negligible compared to the fast execution time of the SMVM and CG algorithms on the GPU. The Prefetch-CSR (PCSR) algorithm proposed in [10] requires very little padding and preprocessing and outperforms the previous SMVM algorithms including one of the best performing algorithms, namely the Row per Warp method from NVIDIA [8]. By using an optimized version of the PCSR algorithm and the Row per Warp method this paper proposes new techniques to overcome major bottlenecks in accelerating PCG on GPUs.

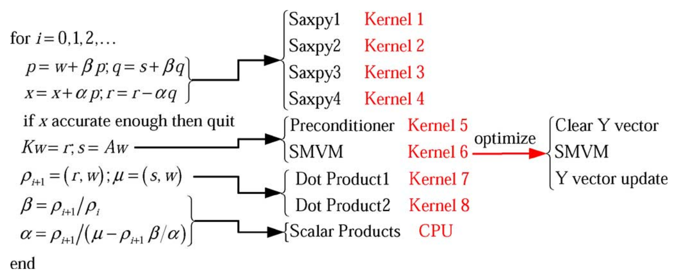

Fig. 3. PCG Chronopoulos [2] algorithm implemented on the GPU, optimizing PCSR [10] adds two new kernels to the implementation.

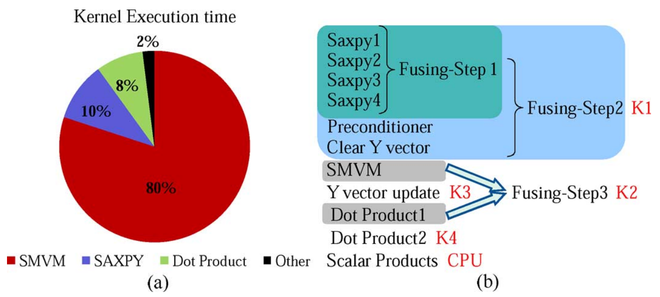

Fig. 4. (a) Percentage of the average execution time of kernels in the PCG Chronopoulos; (b) fusing kernels in PCG (K1 to K4 represent the kernels in optimized PCG).

IV. IMPLEMENTING PCG ON GPUS

We propose four optimizations to the original Choronopoulos PCG in order to decrease its execution time on the GPUs. Without optimization, implementing the Chronopoulos variant of the PCG algorithm leads to 8 kernels and some scalar updates on the CPU (Fig. 3). Fig. 4(a) shows the percentage of average time spent on each of these kernels in the naive implementation of the PCG algorithm on the GPU. We enhance the performance of the Chronopoulos PCG by optimizing the SMVM kernel, fusing SAXPY operations, using a Jacobi preconditioner and binding vectors to GPU texture memory.

A. *Optimizing the SMVM Kernel*

As shown in Fig. 4(a) on average 80% of the total PCG execution time is spent on the SMVM kernel; thus, using the best performing SMVM algorithm is essential in decreasing PCG execution time. In this paper, we compare the effects of two of the best performing SMVM algorithms proposed in previous work [8], [10] in the PCG algorithm. The first algorithm is the Row per Warp method introduced by Bell *et al.* [8] and the prefetch compressed row storage (PCSR) [10] is the second SMVM method used. Unlike SMVM algorithms based on other storage formats, the Row per Warp method and PCSR do not require extra preprocessing since they are based on the compressed sparse row (CSR) storage format.

Row Per Warp: In the Row per Warp [8] method each warp is assigned a row to compute one vector result. The method is efficient if the sparse matrix has a regular sparsity pattern and its bandwidth is approximately equal to a multiple of a warp size.

Prefetch CSR: The PCSR method [10] partitions the matrix nonzeros to blocks of the same size and distributes them amongst GPU resources. The algorithm pads rows with zeros to increase data regularity and use of parallel reduction techniques. Prefetching data is also used to hide global memory accesses. To further increase the performance of the algorithm, in this work we have eliminated the atomic updates of the Y vector by replacing the original SMVM kernel with three sub-kernels, namely, clear Y vector, SMVM and Y vector update (Fig. 3). Thus, in the optimized version of PCSR, atomic sums of the Y vector values corresponding to partitioned rows between blocks are removed (the two added kernels, clear Y vector and Y vector update are small and have a fast execution time compared to the SMVM kernel).

B. Jacobi Preconditioner

A Jacobi preconditioner was implemented mainly for its ease of parallelization in the PCG method. As an additional benefit, because the solver for this type of preconditioner can be treated as a SAXPY operation, it can be fused with other operations as described in the next section.

C. Fusing Kernels

Although the PCG algorithm is mainly implemented on the GPU in previous work, gathering result vector values and performing vector dot products require going back to the CPU, resulting in multiple kernel calls. In each kernel call data is loaded to fast access GPU shared memory in partitions. Upon termination of a kernel, all data is stored back to the GPU global memory, requiring proceeding kernels to reload data to shared memory before their execution. Thus, besides the launching time of each kernel, increased communication is another major drawback of multiple kernel calls.

There are two objectives of fusing individual kernels, the first is to minimize the number of kernels, saving time between kernel calls; and the second is to take advantage of the vectors loaded into shared memory avoiding double loads. The fusions are done in three steps [Fig. 4(b)]. In the first step the SAXPY kernels are fused in to a single kernel. The second step fuses the preconditioner and clear Y vector kernels in to the SAXPY kernel. The dot product and the SMVM kernels are fused into one kernel in the last step (scalar updates of the dot product are still performed on the CPU).

Fusing reduces the total number of kernels from 8 to 4 in the PCG algorithm. Although optimized implementations of other PCG algorithms might result in small number of kernels, the resulting kernels after fusion in the proposed method have significant implications leading to increased performance:

- Most vectors are only loaded once onto shared memory per iteration.
- Fusing the main operations in the PCG algorithm into one kernel [K1 in Fig. 4(b)] increases coalesced memory fetches reducing global memory accesses.
- Kernels 3 and 4 [K3 and K4 in Fig. 4(b)] are small and do not require large number of memory loads.

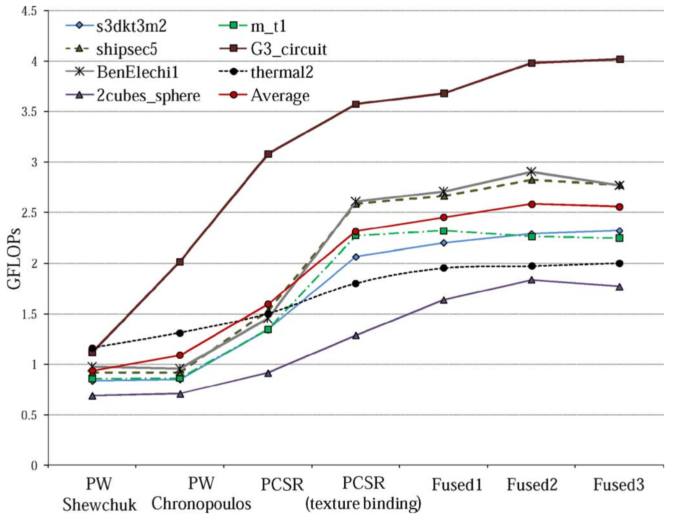

Fig. 5. Effect of the optimizations proposed in Section IV in increasing the performance of the PCG algorithm on GT8800.

TABLE I
SPARSE MATRICES USED FOR TESTING

Matrix Name	Matrix Type	Rows	nnz	nnz/row
thermal2	FEM/steady state	1228045	8580313	7
shipsec5	PARASOL ship	179860	10113096	56
G3_circuit	Circuit simulation	1585478	7660826	5
BenElechi1	2D/3D problem	245874	13150496	53
2cubes_sphere	FEM/sphere	101492	1647264	16
s3dkt3m2	FEM/cyl. shell	90449	3753461	41
m_t1	Tubular joint	97578	9753570	100

D. Texture Binding

The texture memory is a fast on-chip cached memory space. Loading vectors to texture memory decreases the effect of global memory access latencies and enhances the performance of the PCG algorithm kernel. We bind vectors that benefit the most from the cached space to texture memory. By binding vectors to texture memory we increase the execution speed of the PCG algorithm. Since vector values in PCG are updated in each iteration, vectors need to be binded/unbinded to/from texture memory in each iteration.

V. Results

The performance of the optimizations proposed is evaluated using 7 sparse matrices from [11] with different sparsity patterns and application areas (Table I). The execution speed of the PCG algorithm is presented in GFLOPs (billion floating point operations per second). For each PCG Chronopoulos iteration, the algorithm computes 1 SMVM and 7 vector operations; thus, $2 \times nnz + 14 \times n$ flops plus scalar updates are counted (nnz: number of nonzeros, n: matrix dimension).

The performance of the optimized algorithm is tested on two different generations of NVIDIA graphic cards the G80 and GT200 series. NVIDIA GT8800 and GTX280 graphic cards

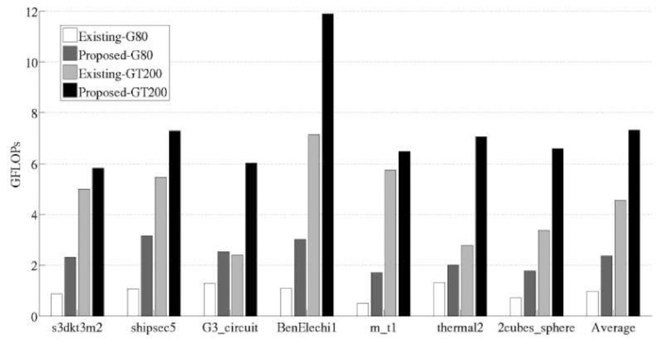

Fig. 6. Performance of the PCG-Row per Warp [8] method compared to proposed optimized PCG Chronopoulos [2] algorithm on G80 and GT200.

are used as representatives of the G80 and GT200 series, respectively. The GTX280 consists of 30 SMs, 16-K registers, and 1 GB of global memory compared to the 14 SMs, 8-K register file, and 512 MB of device memory on the GT8800. Both GPUs have 16 KB of shared memory but the GT8800 operates at a higher frequency (1.5 GHZ versus 1.29 GHZ). The GT200 generation has higher compute capabilities and handles thread divergence more efficiently while the maximum graphic card power and average cost of the GTX280 is approximately double that of the GT8800 card.

Fig. 5 shows the effect of the optimizations proposed in Section IV step by step. Using the Row per Warp algorithm as the SMVM kernel, the PCG Chronopoulos method outperforms the Shewchuk algorithm for all the matrices. By replacing the Row per Warp SMVM with the optimized version of PCSR the average performance of the PCG algorithm increases up to 60% as shown in Fig. 5. While using PCSR as the SMVM kernel, binding vectors to texture memory increases performance on average 50%. Fusing SAXPY operations increases performance on average 6% compared to the nonfused version [Fig. 4(b)]. The two other fusing steps also contribute to an average 6% increase in performance.

Fig. 6 shows the performance of the optimized PCG algorithm compared to the Row per Warp method [8] on both G80 (GT8800) and GT200 (GTX280) NVIDIA GPU generations. The proposed algorithm outperforms previous methods on both platforms. Unlike previous methods [8], [9] which are not optimized for matrices with small number of nonzeros per row, the proposed optimizations, independent of the matrix sparsity pattern, are able to increase considerably the performance for such matrices.

Table II presents the speedup (SU) of the proposed method compared to the Row per Warp (RW) method implemented on the G80 and G200 architectures, the best vectorized CPU results in [12] as well as a naïve CPU implementation. A majority of SMVM algorithms proposed in previous work such as the Row per Warp method introduced in [8] rely on the architecture to address thread divergence; thus, PCG algorithms using such methods do not perform well on the G80 generation of NVIDIA GPUs. Since PCSR's performance is independent of the GPU generation, our PCG implementation outperforms the PCG version of the Row per Warp method on both GPU generations (Table II). Compared to vectorized PCG [12] using 4 threads on an Intel core2 Quad 2.4-GHZ architecture (4 MB of L2 cache per core-pair and 4 GB of global DRAM) speedups of up to 23 were achieved (Table II). On average 42 times speedup was achieved compared to nonvectorized PCG using a single thread on the same CPU ("CPU Regular" results in Table II).

TABLE II
SPEEDUP OF OPTIMIZED PCG COMPARED TO PCG-ROW PER WARP (RW) ON GPU, VECTORIZED AND NONVECTORIZED CPU

Overall Speedup	RW G80	RW GT200	Quad-Core Vector Processing	CPU Regular
s3dkt3m2	2.7	1.16	11.11	30.65
shipsec5	2.97	1.33	14.02	72.92
G3_circuit	1.95	2.49	13.25	26.18
BenElechi1	2.79	1.66	23.32	56.65
m_t1	3.4	1.12	7.45	34.12
thermal2	1.52	2.53	9.39	41.22
2cubes_sphere	2.5	1.95	11.99	31.35
Average	2.55	1.75	12.93	41.87

Compared to single GPU results in [9] (their method uses an SMVM decision algorithm to choose the best performing storage format for each matrix, increasing preprocessing time), for the same matrices we achieve on average 1.5 times speedup (for similar matrices G3_circuit, thermal2, and BenElechi1 speedups of 1.5, 2, and 1.1 are achieved, respectively). Thus, the proposed PCG optimizations can, potentially, give average performances of up to 180 GFLOPS on multi-GPU platforms compared to 120 GFLOPS in [9].

VI. Conclusion and Future Work

The paper introduces several optimizations for the Chronopoulos [2] PCG variant to accelerate the execution of PCG on GPUs. The proposed optimizations increased the performance of PCG on representatives of the G80 and GT200 generations of NVIDIA GPUs up to 3.4 and 2.5 times, respectively, compared to previous methods [8]. In future work, we intend to extend our algorithm to multi-GPU platforms and other preconditioners.

References

[1] L. Buatois, G. Caumon, and B. Lévy, "Concurrent number cruncher: An efficient sparse linear solver on the GPU," in *Proc. HPCC*, 2007, pp. 358–371.
[2] A. Chronopoulos *et al.*, "S-Step iterative methods for symmetric linear systems," *J. Comput. Appl. Math.*, vol. 25, no. 2, pp. 153–156, 1989.
[3] NVIDIA CUDA [Online]. Available: http://developer.nvidia.com/object/cuda.html
[4] J. R. Shewchuk, An Introduction to the Conjugate Gradient Method Without the Agonizing Pain, Aug. 1994.
[5] W. A. Wiggers *et al.*, "Implementing the conjugate gradient algorithm on multi-core systems," in *Proc. ISSC*, 2007, pp. 11–14.
[6] S. Georgescu and H. Okuda, "GPGPU-enhanced conjugate gradient solver for finite element matrices," presented at the iWAPT, 2007.
[7] D. Goddeke, R. Strzodka, and S. Turek, "Accelerating double precision FEM simulations with GPUs," presented at the ASIM, 2005.
[8] N. Bell and M. G. Fernandez, Efficient Sparse Matrix-Vector Multiplication on CUDA, NVIDIA Tech. Rep., 2008.
[9] A. Cevahir, A. Nukada, and S. Matsuoka, "High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning," *J. Res. Develop.*, vol. 5, no. 1, pp. 83–91, 2010.
[10] M. M. Dehnavi, D. Fernandez, and D. Giannacopoulos, "Finite element sparse matrix vector multiplication on GPUs," *IEEE Trans. Magn.*, vol. 46, no. 8, pp. 2982–2985, Aug. 2010.
[11] University of Florida Sparse Matrix Collection [Online]. Available: http://www.cise.ufl.edu/research/sparse/matrices
[12] D. Fernandez, D. Giannacopoulos, and W. J. Gross, "Multicore acceleration of CG algorithms using blocked-pipeline-matching techniques," *IEEE Trans. Magn.*, vol. 46, no. 8, pp. 3057–3060, Aug. 2010.