SpMV: A Memory-Bound Application on the GPU Stuck Between a Rock and a Hard Place

John D. Davis and Eric S. Chung
Microsoft Research Silicon Valley
{*joda, erchung*}*@microsoft.com*

***Abstract*—In this paper, we investigate the relative merits between GPGPUs and multicores in the context of sparse matrix-vector multiplication (SpMV). While GPGPUs possess impressive capabilities in terms of raw compute throughput and memory bandwidth, their performance varies significantly with application tuning as well as sparse input and format characteristics. Furthermore, several emerging technological and workload trends potentially shift the balance in favor towards the multicore, especially in the context of bandwidth-limited workloads. In light of these trends, our paper investigates the performance, power, and cost of state-of-the-art GPGPUs and multicores across a spectrum of sparse inputs. Our results reinforce some current-day understandings of GPGPUs and multicores while offering new insights on their relative merits in the context of future systems.**

I. Introduction

Sparse matrix-vector multiplication (SpMV) is a critical kernel employed in many high performance computing (HPC) applications. SpMV is a dominant component of iterative methods that solve large-scale linear systems and eigenvalue problems. The widespread importance of SpMV has resulted in significant research efforts poured into optimizing the kernel's performance on modern-day multicores and general-purpose GPU (GPGPU) platforms (e.g., [1, 2]). Standard libraries such as Intel's MKL [3] and Nvidia's CUBLAS [4] and Cusp [1] provide users with highly tuned SpMV implementations designed to maximize core efficiency and memory bandwidth.

In this paper, we investigate the relative merits between multicores and GPGPUs when optimizing a commodity system for performance under approximately equal power and cost constraints. A recurring claim that has been made in recent years is that GPGPUs offer significant performance advantages over conventional multicores, especially with respect to SpMV [1]. Proponents highlight the GPGPU's advantage in memory bandwidth and its abundance of floating point cores. For some sparse matrices, GPGPUs achieve over 4X throughput improvement over the multicore in previously conducted studies [1].

Are GPGPUs Worth The Effort? Despite the impressive raw capabilities of GPGPUs, such performance gains are achievable only when extensive tuning efforts are employed and when certain sparse inputs are regular and amenable to load-balancing. Furthermore, we have begun to observe several emerging technological and workload trends that potentially mitigate these benefits: (1) the convergence of memory bandwidth between CPUs and GPGPUs, (2) increasing last-level cache sizes of multicores, and (3) the trends towards large-scale inputs (i.e., "big data").

Contributions. In this paper, we re-examine the GPGPU's advantage over conventional multicores and investigate whether the increased programmability burden and tuning are justifiable in the context of building a system with approximately equal cost and power budgets. To conduct our investigations, we perform detailed performance and power measurements of state-of-the-art platforms using GPGPUs and multicores. From our experimental measurements, we derive several observations:

- For SpMV, GPGPU performance has stagnated in recent architecture generations, despite extensive tuning efforts by the research community.
- In general, GPGPUs achieve high speedups over the multicore on input set sizes that exceed the multicore's last-level cache, but do not exceed the GPGPU's external DRAM capacity.
- SpMV multicore performance has been gaining on the GPU in recent years due to improved memory bandwidth in commodity multicore systems.
- To the first order, a dual-socket multicore is sufficient to replace a single GPGPU assuming approximately equal cost and power consumption.
- When extrapolated to future technological trends, the advantages of using GPGPUs for SpMV diminishes significantly as multicores are able to catch up in bandwidth-limited performance.

Outline. The remainder of this paper is as follows. Section II discusses key technological trends that motivate this work. Section III describes background and related work. Section IV elaborates on the experimental setup used to conduct our studies. Section V discusses in detail the experimental results. Section VII offers conclusions and future work.

II. Motivation and Trends

Over the last several years, researchers and application developers have poured an immense amount of effort into the programmability, tuning, and demonstration of general-purpose graphic processors (GPGPUs). In contrast to conventional multicores, GPGPUs offer substantial peak computational throughput (on the order of teraflops/sec) and high memory bandwidth (nearly up to 200 GB/sec). GPGPUs achieve good performance and efficiency by relying on hundreds, if

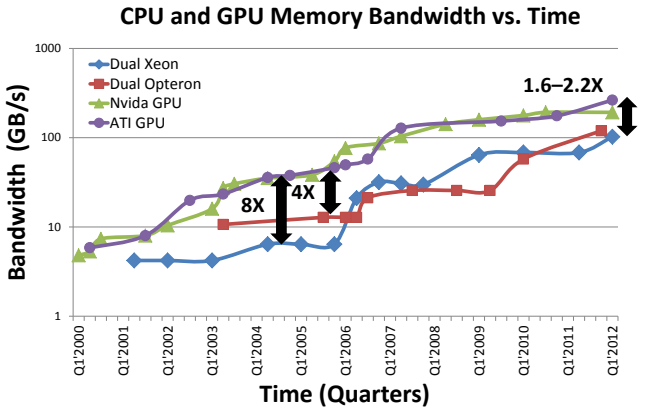

Fig. 1. Bandwidth Gap between GPGPUs vs. CPUs.

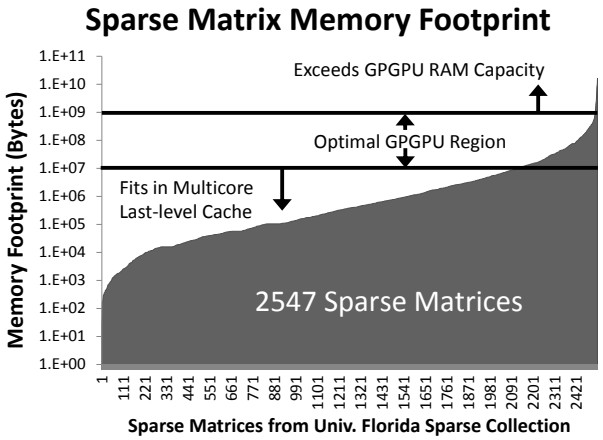

Fig. 2. Sparse Matrix Memory Footprint.

not thousands of threads to extract parallelism from a target application.

Recently, emerging technological trends are calling into question as to whether GPGPUs are "always the right answer" when high performance is of utmost demand. Unlike conventional processors, GPGPU applications are considerably fragile and require extensive tuning to maximize effectiveness. In the case of Sparse Matrix-Vector Multiplication (SpMV), we are particularly interested in knowing whether well-tuned GPGPUs today (and moving forward into the future) will provide a significant enough performance (or energy) benefit to justify their complex tuning and programming requirements.

Bandwidth Trends. A potential threat to the GPGPU's dominance is the fact that conventional multicores are closing the gap in performance and bandwidth. As Figure 1 shows, the memory bandwidth gap between GPGPUs and multicores has been shrinking over the last five years. As recently as 2012, a high-end dual-socket Xeon system can achieve over 100 GB/sec of memory bandwidth, a level difficult to envision five years ago.

The remaining advantage of the GPGPU over CPU is now the raw processing throughput that compute-bound applications can exploit. However, in many memory-bound applications such as SpMV, it becomes increasingly unclear whether GPGPUs can still offer a significant advantage. In particular, SpMV is classified as an irregular memory access application, which must be optimized along the following axes: (1) memory efficiency, (2) load balancing, and (3) thread utilization. This requires careful mapping and data marshaling to support the desired computational throughput. Although conventional multicores suffer similar problems, the developer is responsible for managing far less complexity than in the GPGPU (tens of threads vs. thousands). Furthermore, conventional multicores also contain large last-level caches (20MB or more) that automatically improve memory efficiency.

Memory Capacity Trends. In addition to increased bandwidth, the last-level caches of multicores continue to grow unabated. The Intel Sandy Bridge architecture [5] enables up to 20MB of on-die cache, or 40MB in a dual-socket Xeon configuration [5]. The successor architecture, Ivy Bridge, can expect up to 32MB of last-level cache per die. In contrast, an Nvidia GPGPU provides only a scant amount of cache, on the order of a megabyte or less [6]. An interesting consequence of having massive caches in the multicore is the impact on traditional "memory-bound" iterative solvers. Across a collection of 2,547 real-world sparse matrices from the University of Florida Sparse Collection [7], 93% of the matrices can fit entirely in the 40 MB cache (not including metadata overhead), which ultimately negates the benefit of having extremely high external bandwidth, as in the case of the GPGPU. Figure 2 plots each of the individual matrices against the total memory footprint. The lower horizontal bar indicates the last-level cache size of a Sandy Bridge-based Dual-Socket Xeon, while the upper horizontal bar indicates the maximum GDDR capacity of an Nvidia GTX680 GPGPU.

Long-Term Hypothesis. Based on Figure 2, we project that future "big data" trends will ultimately limit the utility of running SpMV on GPGPUs, which must either constrain their inputs to limited off-chip memory or must reload their data over a slow PCIe bus, destroying any of the benefits that GPGPUs may provide. Furthermore, the favorable regions where GPGPUs can excel at will be gradually subsumed by more capable CPUs, while larger data sets will push such workloads out of reach of the memory-constrained GPGPUs[1]. In the next section, we will provide background knowledge on the SpMV kernel and various CPU and GPGPU implementations. This section is followed by an experimental study used to test our long-term hypothesis.

III. Sparse Matrix-Vector Multiplication

Sparse Matrix-Vector Multiplication is a highly exercised scientific kernel that solves $y = Ax$, where x and y are dense vectors, while A is a sparse matrix. In iterative methods, the SpMV kernel is typically executed thousands of times before convergence is reached. To avoid excessive waste in bandwidth

[1] *Or stuck between a rock and a hard place.*

and storage, the sparse matrix A is typically encoded using a **sparse format**, which stores only the non-zero values of the matrix along with meta-data that identifies a non-zero's location in the matrix.

Using the right sparse matrix format is essential because at runtime, the metadata must be decoded for each non-zero, which is used to identify the corresponding x vector value for calculating the dot product. A major challenge in computing $y = Ax$ is the irregularity of accesses to the x vector as a result of the non-zero access patterns in the matrix.

It is notable that GPGPUs are very sensitive to the sparse matrix format. The format and resulting metadata can impact the load balance, resource utilization, and memory efficiency. As Bell and Garland showed, an "improper" sparse matrix format can degrade performance by an order of magnitude or more [8].

Sparse Formats. A wide variety of sparse formats exist, trading storage/bandwidth and processing overhead. Coordinate (COO), Compressed Sparse Row (CSR), and ELLPACK (ELL) are common formats supported by standard sparse matrix packages like SPARSKIT [9]. Figure 3 illustrates these formats for an example sparse matrix. The COO format is the simplest, where each row and column index is stored along with the non-zero value. In the CSR format, the row array is replaced by pointers that demarcate the beginning of rows within the column array. Finally, in the ELL format, the entire sparse matrix is packed into a dense matrix with dimension k, where k is the largest number of non-zeros amongst all the rows. A secondary dense matrix encodes the column index for each respective nonzero. ELL format is most well-suited for vector architectures because of its fixed length rows.

Table I provides a summary of revelant details of some of the matrices used in this and other studies. We also profiled the entire University of Florida Sparse Matrix Collection to understand the storage ramifications of these popular formats. On the right of the table, we normalize the storage requirements to the most common format, CSR. In general, COO is the most stable format compared to CSR, having low overhead and a low standard deviation. For matrices with a uniform number of nonzero values, ELL can be the most storage efficient, because there is no need to store row metadata. It also can experience massive amounts of overhead due to data and metadata padding for matrices with very skewed nonzero distributions. Finally, Bell and Garland derived their own format, a combination of ELL for the bulk of the matrix and COO to store the remaining rows with a large number of nonzeros [8]. On average, this format sacrifices storage efficiency relative to CSR and COO, with a focus on improving GPGPU performance.

Multicore Optimizations. A substantial body of literature has explored the optimization of sparse formats and algorithms for single-core and multi-core processors (e.g., [10, 11, 2, 9, 12, 13]). In general, many optimizations aim to minimize the irregularity of the matrix structure by selecting a format best suited for the matrix kernel. Commercial libraries such as Intel MKL provide optimized SpMV libraries for Intel-based processors. In processor-based systems, multithreading, SSE instructions, cache blocking and mitigation of x-vector irregularity are common techniques used to boost the efficiency and performance of SpMV. The use of auto-tuning has also been shown to be an effective technique for selecting good optimizations for a given processor's configuration and platform [2].

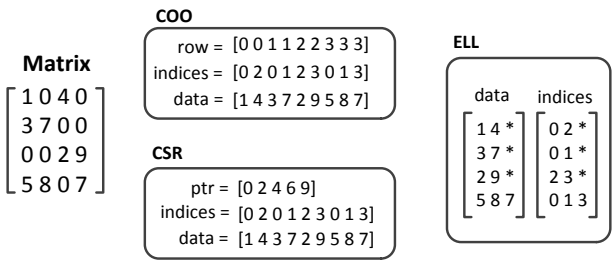

Fig. 3. Various Formats Used to Encode Sparse Matrices.

GPGPU Optimizations. In the space of GPGPUs, a significant body of work exists on creating specialized data formats that mitigate load-balance issues. The performance of SpMV on GPGPUs is particularly sensitive to the format used to encode the matrix. An ELL format, for example, presents a simple array-based structure that simplifies load-balancing and scheduling on the GPGPU. However, ELL also incurs substantial waste if there is significant variability in the number of non-zeros per row [1]. A corollary to the format's performance is its storage and bandwidth requirements. As shown in Table I, the storage requirements on the GPGPU can also impact performance. In our evaluations that compare the GTX680 and GTX580, there existed several large matrices that fit in the memory of the GPGPUs in one format, but not the other. In some cases, the GTX680 was able to exploit a given format because of its large memory, whereas the GTX580 could not. The same holds true when comparing CPU and GPGPU performance. We will later present a few examples of the CPU's performance being artificially inflated because the GPGPU does not have the memory capacity for the faster format.

Related Work. Related to CPUs and GPGPUs, we build on the auto-tuning results of Williams et al. [2] and the cusp library as discussed by Bell and Garland [8]. We extend their work based on the narrowing bandwidth gap and investigate newer CPU and GPU architectures. We measure both performance and power using the last version of their code [1] and across a spectrum of inputs that vary the impact of the last-level cache [7]. Likewise, Lee et al. [14] presents performance results demonstrating the narrowing performance gap across a wide range of workloads, but omit any power comparisons. From this work and our experience, tuning and debugging the GPGPU is significantly more challenging than CPUs. In particular, we noticed a significant amount of variability in performance with respect to the format and data set size.

	matrix	rows	cols	nonzeros	% nonzeros	CSR	COO	ELL	HYB
	dense	32,768	32,768	1073741824	100 %	1.0	1.33	1.00	1.50
Inputs used in [8]	cant	62451	62451	4007383	0.10%	1.0	1.33	1.21	1.75
	conf5_0-4x4(qcd)	3072	3072	119808	1.27%	1.0	1.32	0.99	1.32
	conf5_4-8x8 (qcd)	49152	49152	1916928	0.08%	1.0	1.32	0.99	1.32
	consph	83334	83334	6010480	0.09%	1.0	1.33	1.12	1.82
	cop20k_A	121192	121192	2624331	0.02%	1.0	1.31	3.68	1.99
	epb1	14734	14734	95053	0.04%	1.0	1.27	1.03	1.54
	mac_econ_fwd500	206500	206500	1273389	2.99e-3%	1.0	1.26	6.77	1.64
	mc2depi	525825	525825	2100225	7.60e-4%	1.0	1.23	0.92	1.92
	pdb1HYS	36417	36417	4344765	0.33%	1.0	1.33	1.71	1.82
	pwtk	217918	217918	11634424	0.02%	1.0	1.33	3.35	1.53
	rail4284	4284	1096894	11284032	0.24%	1.0	1.33	21.3	1.34
	rma10	46835	46835	2374001	0.11%	1.0	1.32	2.84	1.79
	scircuit	170998	170998	958936	3.28e-3%	1.0	1.26	59.4	1.52
	shipsec1	140874	140874	**3977139***	0.04%	1.0	1.33	1.83	1.66
	webbase-1M	1000005	1000005	3105536	3.11e-4%	1.0	1.20	1366	1.49
All sparse matrices [7]	Mean	361922	361132	4448472	1.82%	1.0	1.26	390	1.46
	Median	4182	5300	40424	0.23%	1.0	1.27	2.77	1.31
	Stdev	3339878	3333546	55947269	6.10%	1.0	0.06	6092	0.32
	Min	2	3	3	2.09e-6%	1.0	0.85	0.55	0.85
	Max	118142142	118142155	1949412601	76.0%	1.0	1.33	250108	3.69

TABLE I
CHARACTERIZATION OF ALL SPARSE INPUT MATRICES FROM THE UNIVERSITY OF FLORIDA SPARSE MATRIX COLLECTION [7]. STORAGE VALUES NORMALIZED TO CSR FORMAT.

		Intel Bloomfield	Intel Sandy Bridge-EP	Nvidia GTX580	Nvidia GTX 680
Technology	*Year*	2008	2012	2010	2012
	Node	Intel/45nm	Intel/32nm	TSMC/40nm	TSMC/28nm
	Clock rate	3.2GHz	2.4GHz	1.5GHZ	0.7-1GHz
	Die area	$263mm^2$	$416mm^2$	$529mm^2$	$296mm^2$
	Transistors	763M	2.26B	3B	3.54B
	Cores/Threads	4/8	8/16	32/512	8/1536
	Last-level Cache	8MB	20MB	768kB	512kB
	DRAM	16GB	32GB	1.5GB GDDR5	2GB GDDR5
	Bandwidth	25.6GB/sec	51.2GB/sec	192.4GB/sec	192.4GB/sec
	List Price	$300	$1400	$500	$500

TABLE II
SUMMARY OF DEVICES.

IV. EXPERIMENTAL SETUP

Our experiments target multiple hardware platforms across CPUs and GPUs in the context of SpMV. The target platforms share similar silicon technology feature sizes and utilize optimized software libraries. In this section, we briefly describe the architectural features of the systems: (1) the single-socket quad-core Intel Xeon W3550, (2) the dual-socket × 8-core Intel Xeon E5-2665, (3) the Nvidia GTX580 and (4) the Nvidia GTX680. We report measured full-system power using a digital power meter with measurements collected once per second in all cases. Table II provides a summary of all the platforms used in this study.

Intel Bloomfield. We include the Nehalem-based, Intel Xeon Bloomfield (W3550) processor as a comparison point for the work published by Bell and Garland [8]. They used a similar Core i7 965 processor for their comparison. For consistency, we include this platform as a baseline for the SpMV workload. This processor has four cores operating at 3.2 GHz and supports 2 hyperthreads per core or up to 8 threads in the system. It has an 8 MB last level cache and three memory channels with a maximum memory bandwidth of 25.6 GB/s. This processor supports SSE4.2, enabling the use of SIMD instructions in our CPU evaluation. This processor is manufactured using a 45 nm silicon technology with 763 million transistors on a 263 mm^2 die and is the largest feature size device used in our study. This processor has a list price of about $300 [5]. We did collect multicore performance numbers from this platform as a sanity check against the published results. Our performance results exceeded those results of the Intel Core i7 965 CPU in [8] and are not presented in this paper.

Intel Sandy Bridge-EP. We use a more current Intel Xeon Sandy Bridge-EP processor in a dual-socket system as an example of a similar system compared to a workstation or single-socket system which includes a GPGPU. This processor has 8 cores operating at 2.4 GHz and supports 2 hyperthreads per core or up to 16 threads per socket. Each core has a 20 MB last level cache and four memory channels with a maximum memory bandwidth of 102.4 GB/s for the dual socket system. We are using DDR3-1333 DRAM DIMMs, so the system's peak memory bandwidth is limited to 85.3 GB/s. This processor supports SSE4.2 and SIMD instructions are used in our evaluation of the multicore performance. This

processor is manufactured using 32 nm silicon technology with 2.26 billion transistors on a 416 mm^2 die. This processor has a list price of about $1400 [5].

Nvidia GTX580 (Fermi). The first GPU we use is the Fermi-based Nvidia GTX580. This GPU has 16 streaming multiprocessors with 32 CUDA processor each, for a total of 512 CUDA cores running at 1.5 GHz. The GTX580 has a 32K x 32-bit register file and a 64 KB shared L1 cache. There is also a 768 KB L2 cache. This particular card has 1.5 GB of GDDR5 memory, with a 384-bit wide memory interface, operating at 2.0 GHz for a maximum memory bandwidth of 192.4 GB/s. This GPU is manufactured using 40 nm silicon technology with 3 billion transistors on a 520 mm^2 die. This GPU was purchased for about $500.

Nvidia GTX680 (Kepler). We compare the GTX580 to the latest Kepler-based Nvidia GTX680. This GPU has 1536 CUDA cores running at 700 MHz, but can overclock to 1 GHz on high loads. These CUDA cores are grouped into eight streaming multiprocessor with 192 CUDA cores. The GTX680 has a 64K x 32-bit register file and a 64 KB shared L1 cache. There is also a 512 KB L2 cache, far less memory per thread than the GTX580. This particular card has 2.0 GB of GDDR5 memory, with a 256-bit wide memory interface, operating at 3.0 GHz for a maximum memory bandwidth of 192.2 GB/s, maintaining the same memory bandwidth as the GTX580. This GPU is manufactured using 28 nm silicon technology with 3.54 billion transistors on a 296 mm^2 die. This GPU was purchased for about $500.

A. Software and Measurements

Nvidia Cusp. The Nvidia Cusp library provides a collection of basic linear algebra subroutines (BLAS) for sparse matrices. The implementation of SpMV in Cusp is based on optimizations by Bell and Garland [1], which utilize the hybrid format (HYB) as discussed in Section III. We use a new and improved version of the Cusp library, v0.3.0, with Thrust v1.5.2 and CUDA 4.2. We compile the GTX580 binaries with the $arch = sm\_20$ flag. Likewise, the target binaries for the GTX680 are compiled with the $arch = sm\_30$ flag. The GPUs are installed in an HP Z400 workstation running 64-bit Windows 7 Enterprise Edition, as well as a 1U XEON-based server running Windows Server 2008 R2. There was no significant performance difference of the same GPU running on the different platforms.

We use the same test harness that Bell and Garland [1] used for their study, but with a superset of input matrices to investigate performance constraints related to device and system memory. The Cusp library has also been improved since the initial paper, providing a slight advantage over the CPU code and environment used in this study. The test harness converts the Matrix Market format into one of five formats: COO, CSR, DIA, ELL, and HYB. Furthermore, explicit cache management is another dimension the user can control. We found that COO and scalar CSR never provided the best performance, so these formats were excluded from the power measurement phase. For each input matrix, we selected a subset of the available formats (CSR vector, DIA, ELL, HYB) for measurement and ran at least 10,000 iterations or 10 seconds to faciliate stable power measurement. We report the best average results observed over several repeated runs. The initialization and data transfer times are not measured to be consistent with prior work [2, 8].

Auto-tuned CPU Code. We utilize the sparse auto-tuning framework from Williams et al. [2] to generate an optimized SpMV solution for each sparse matrix. The library is multi-threaded and explores optimizations such as cache and TLB blocking to maximize efficiency of all the cores. This code was ported to run in Cygwin 1.7, using gcc 4.5.3. Porting the code required removing any 64-bit references in the toolchain, adding memory allocation alignment code, and removing references to the Linux thread scheduler. The dual-socket Xeon server ran with hyperthreading enabled. We did not observe a significant degradation in performance when hyperthreading was disabled. The server was running Windows Server 2008 R2 SP1.

In order to produce repeatable results, we use an initial short run to determine the optimal configuration for each input matrix. We then restrict the search space of the auto-tuner to the settings that provided the best performance and re-ran the SpMV kernel with that particular input matrix for 10,000 iterations or at least 10 seconds to facilitate stable power management. We do not restrict the blocking algorithm. We report the maximum performance and its associated full-system power measurement for each input matrix. As with the GPGPU results, the performance measurements neglect format conversion time and focus on measuring the kernel runtime.

Power Measurement. We measure full-system power using the *watts up? Pro* digital power meter. We measure and log full system power with a frequency of 1 Hz. We use the Joulemeter framework [15] to register the power performance counter and the ETW framework [16] to record the full system power for all the experiments. We report the average power measured over the interval of each run. When using the GPGPU, we observe the transistion points between formats and can demarcate regions of interest. Likewise, the many-core SpMV kernel exhibited three distinct and identifiable auto-tuning phases.

V. Results

Performance Comparison. Figure 4 compares the performance of SpMV across various sparse inputs and devices. Each bar in Figure 4 shows the performance of either the GTX680 or dual-socket Xeon normalized to the GTX580's performance. For each of the inputs, the x-vector size grows from left-to-right. The performance bars are further separated into two categories: in-cache and out-of-cache with respect to the dual-socket Xeon (i.e., 40MB LLC).

From initial observation, it can be seen that the GTX680 *underperforms* the GTX580 for nearly all of the inputs, which is surprising given that the GTX680 is a newer architecture

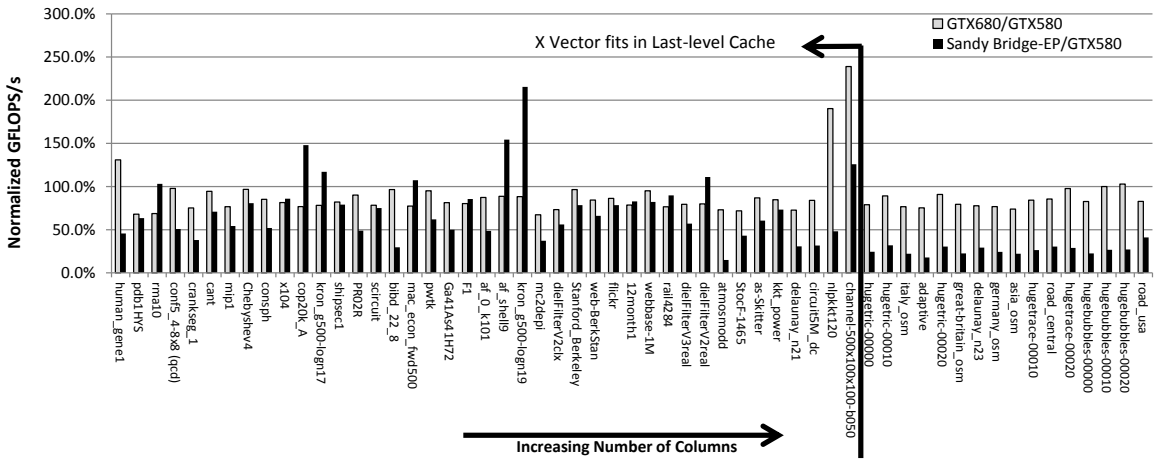

Fig. 4. Performance of Sparse Matrices Across GTX580, GTX680, and Sandy Bridge-EP.

with many more cores. Two possible reasons for this disrepancy are: (1) lack of tuning of the Cusp library beyond using the correct compilation flags, and/or (2) the effect of Kepler having a **smaller** last-level cache and overall greater memory pressure compared to Fermi, which would increase the miss rate and hence delay accesses to the x-vector elements.

A key observation we make in Figure 4 is that the GTX580/GTX680 speedup over the multithreaded Xeon implementation is approximately 2X only in the in-cache regime, highlighting the significance of the Xeon's massive last-level cache (40MB). Recall from Section III that nearly 93% of all the sparse inputs from the University of Florida Sparse Collection would fit in such a cache size. This percentage increases to 99% if we only consider the most critical structure, which is the x vector. The left-hand side of the figure is sampled from the 99% portion of the matrices, where the x vector does fit in the last-level cache. In this regime, the GPGPU does not offer a significant advantage, especially given the significant amount of tuning involved.

The right-hand-side of Figure 4 illustrates the out-of-cache inputs, representing about 1% of the matrices in the University of Florida matrix repository [7]. In this regime, the GTX580/GTX680 achieves on average 3X speedup over the Xeon due to their higher external memory bandwidths. However, these speedups are relatively modest for only a small subset of the inputs, and a significant portion of these inputs do not fit in GPGPU's off-chip memory, reducing potential candidates even further.

Finally, Section III described GPGPU sensitivity to the matrix format. The two inputs to the left of the vertical line (nlpkkt120 and channel-500) demonstrate this sensitivity. Both matrices have x vectors that fit in the aggregate last-level cache (40 MB), but the overall matrix and metadata for some formats do not fit in the GTX580 memory (1.5 GB vs 2.0 GB for the GTX680). For channel-500, both the Xeon and GTX680 outperform the GTX580, even though the GTX680's performance is lower than the GTX580 using the same input format. The same holds true for the nlpkkt120 matrix. Again, formats being equal, the GTX580 outperforms the GTX680, but because the GTX680 has more memory, it can use a format that yields higher performance.

Power and Energy efficiency. Figure 5 shows the full-system power consumption of the GTX580 and GTX680 in the single-socket host and the dual-socket Xeon-based server. In general, the power consumption does not vary significantly between inputs. Across the platforms, the GTX680 is slightly more efficient and consumes on average about 15% less power than the GTX580, achieving comparable power consumption to the Xeon-based system. Figure 6 further normalizes the performance to power, which computes the number of operations per Joule of energy. Unsurprisingly, the efficiency characteristics follow a similar trend to performance for the CPU compared to the GPGPUs, i.e., comparable efficiency with in-cache inputs vs. at least 2X efficiency improvement in the out-of-cache regime. Surprisingly, across the set of sparse matrix inputs, the GTX680 is not more efficient than the GTX580 given its lower TDP and smaller silicon process technology. We address this observation below.

GPGPU Area Efficiency. Figure 7 normalizes the performance of each of the inputs to the technology-normalized die area (28nm) of each respective device. We essentially scale down all of the devices to the same silicon technology and compare the MFLOPS/Joule. When normalizing area for silicon technology scaling, the scaled-down size of the GTX580 would actually be smaller than the GTX680. Thus, the GTX580 has slightly better efficiency due to its smaller normalized die area and slight performance edge. Figure 7 clearly shows that GPGPUs are more area efficient compared to CPUs, demonstrating their merits in the context of a single-chip heterogenous die. As can be seen, both the GTX580 and GTX680 achieve significant gains in area efficiency relative to the Xeon-based multicore, which requires significant area for the large, last-level caches. In a power and/or energy-constrained environment, utilizing area for the GPGPU is a plausible design choice in the context of building accelerators

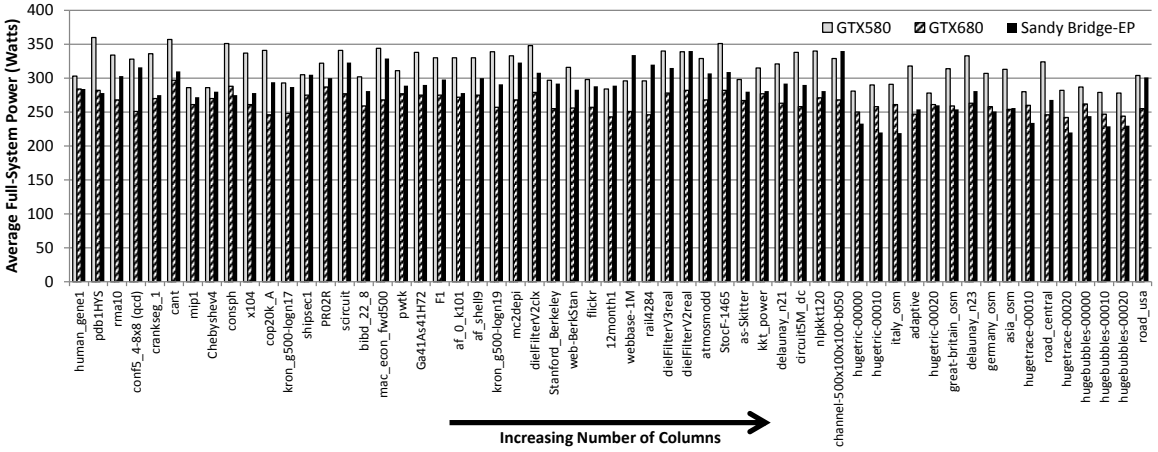

Fig. 5. Measured Power Across GTX580, GTX680, and Sandy Bridge-EP.

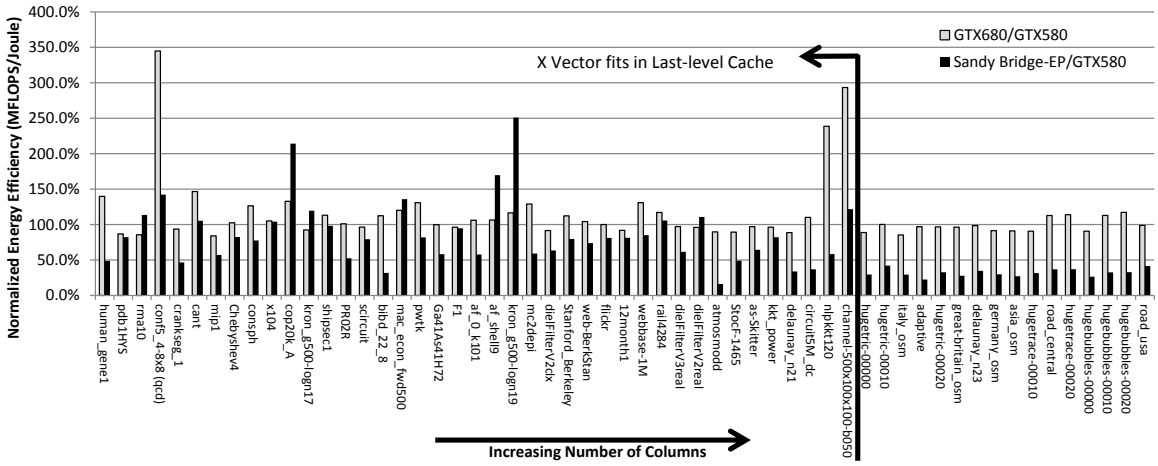

Fig. 6. Measured Energy Efficiency Across GTX580, GTX680, and Sandy Bridge-EP.

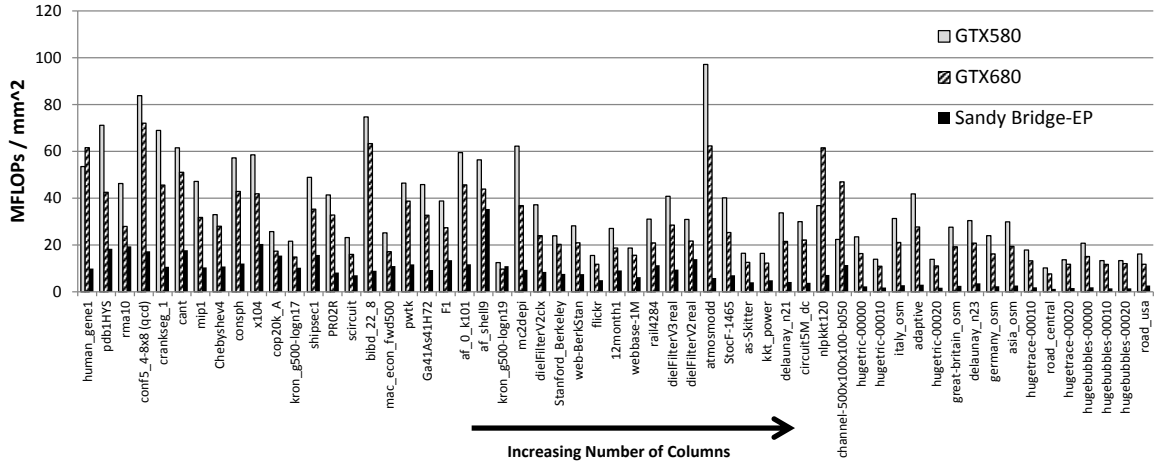

Fig. 7. Area-Normalized Performance Across GTX580, GTX680, and Sandy Bridge-EP.

in an energy-constrained environment.

VI. Discussion and Future Work

This work ushers in the first major GPU release where improvements in silicon technology do not translate directly into specification and application improvements. Triggered by this event, we investigated the relative merits between GPGPUs and multicores in the context of sparse matrix-vector multiplication (SpMV). While GPGPUs possess impressive capabilities in terms of raw compute throughput and memory bandwidth, their performance varies significantly with application tuning as well as sparse matrix input and format characteristics.

While XDR is poised to replace GDDR memory in the near future, this gap will likely remain constant given that both CPU and GPGPU vendors have access to this technology. Furthermore, several emerging technological and workload trends potentially shift the benefit towards the multicore, especially in the context of bandwidth-limited workloads. Looking further into the future, in-package or on-die integration of CPUs and GPUs represents a scenario where off-chip memory bandwidth is shared or the same, which brings the issue of platform selection to the forefront of system design. This raises a critical question for computer architects on how to allocate silicon real estate between core types. Currently, their choices range from favoring area efficiency by using more area for GPGPUs or using conventional cores to retain programmability. We believe GPU and CPU designers are now up for determining the right mixture of components in the future.

In addition to technological trends, "Big data" is becoming the norm, which pushes working set sizes outside the reach of iterative solvers mapped to memory-limited GPUs. Furthermore, the increased number of cores per die, memory bandwidth and growing last-level caches is beginning to erode the performance benefit of GPGPUs relative to CPUs (although not their energy efficiency). Our experiments show that GPGPU only offers significant benefit for certain classes of inputs that (1) do not exceed the available external GDDR capacity, and (2) are sufficiently large in memory footprint (i.e., significantly larger than a multicore's last-level cache). On average, for the SpMV inputs we used, the GTX680 was 15% slower (GFLOPS) compared to the GTX580. Furthermore, the Xeon-based server was 25% slower (GFLOPS). However, when intelligent caching was not an option for the CPU, its performance was nearly 75% lower.

There are several possible solutions to prevent further stagnation. Like the Xeon-server, the GPGPUs could be aggregated seamlessly in multi-socket server configurations. Creating a unified environment, although desirable, would not be trivial due to issues such as coherency and memory. GPGPUs could also benefit from larger shared caches or at least maintain a constant amount of memory per thread. Finally, in the short term, DRAM bandwidth can be improved dramatically using technologies such as Rambus XDR, although this will not stop the impending memory bandwidth convergence, especially if high-end GPUs and CPUs share the same package.

In the future, we would like to further investigate the role of custom accelerators beyond GPGPUs (e.g., ASICs, FPGAs) in this domain to understand their role and where they fit in Figure 2. We would also like to test our hypothesis on other applications such as machine learning or other sparse iterative methods.

VII. Conclusions

In the context of memory-bound applications such as Sparse Matrix-Vector Multiplication, GPGPUs are "stuck" between a rock and a hard place. From one direction, modern multicores have been rapidly gaining in last-level cache sizes and external memory bandwidth, which poses a threat to the GPGPU. From the other direction, data set sizes are continuing to grow unabated in the era of "big data", yet stand-alone GPGPUs remain limited by DRAM capacity. These emerging trends pose an even greater barrier to adoption, especially given the wide gap in programmability between a CPU and GPU. Despite these trends, our experimental results do show that GPGPUs are extremely area-efficient, which suggests that GPGPUs are excellent building blocks for future, single-chip heterogeneous processors. In the future, we plan to extend our studies to incorporate other memory-bound applications.

Appendix

For reference, Figure 8 shows all of the raw data collected in our experiments.

References

[1] N. Bell and M. Garland, "Cusp: Generic parallel algorithms for sparse matrix and graph computations," 2012, version 0.3.0. [Online]. Available: http://cusp-library.googlecode.com

[2] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *In Proc. SC2007: High performance computing, networking, and storage conference*, 2007, pp. 10–16.

[3] "Intel Math Kernel library." [Online]. Available: http://software.intel.com/en-us/intel-mkl

[4] "Nvidia cublas."

[5] "Intel Processor Information." [Online]. Available: http://ark.intel.com/

[6] "GeForce Graphics Processors." [Online]. Available: http://www.nvidia.com/object/geforce_family.html

[7] T. A. Davis, "University of Florida Sparse Matrix Collection," *NA Digest*, vol. 92, 1994.

[8] N. Bell and M. Garland, "Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC'09. New York, NY, USA: ACM, 2009, pp. 18:1–18:11.

[9] Y. Saad, "SPARSKIT: a basic tool kit for sparse matrix computations - Version 2," 1994.

[10] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 135–158, Feb. 2004. [Online]. Available: http://dx.doi.org/10.1177/1094342004041296

[11] R. Vuduc, J. Demmel, K. Yelick, S. Kamil, R. Nishtala, and B. Lee, "Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply," in *Supercomputing, ACM/IEEE 2002 Conference*, nov. 2002, p. 26.

	GTX580			GTX680			Sandy Bridge-EP		
name	GFLOPS	Power	MFLOPS/J	GFLOPS	Power	MFLOPS/J	GFLOPS	Power	MFLOPS/J
human_gene1	13.9	303	45.8	18.2	284	63.9	6.3	284	22.3
pdb1HYS	18.4	360	51.2	12.5	282	44.5	11.7	278	42.1
rma10	12.0	334	35.9	8.2	268	30.7	12.4	303	40.8
conf5_4-8x8 (qcd)	21.7	328	24.5	21.2	251	84.6	11.0	316	34.9
crankseg_1	17.9	336	53.2	13.5	270	49.8	6.8	275	24.7
cant	15.9	357	34.6	15.1	297	50.7	11.3	310	36.5
mip1	12.2	286	42.7	9.4	261	35.9	6.6	272	24.4
Chebyshev4	8.5	286	29.9	8.3	270	30.6	6.9	280	24.6
consph	14.8	351	36.1	12.6	288	45.7	7.7	275	28.0
x104	15.2	337	45.0	12.4	261	47.3	13.0	278	46.9
cop20k_A	6.7	341	15.7	5.1	246	20.8	9.9	294	33.6
kron_g500-logn17	5.6	293	19.1	4.4	248	17.7	6.6	287	22.8
shipsec1	12.7	305	33.5	10.4	275	37.8	10.0	305	32.8
PR02R	10.7	322	33.3	9.7	287	33.7	5.2	300	17.5
scircuit	6.0	341	17.6	4.7	277	17.0	4.5	323	14.0
bibd_22_8	19.4	302	64.1	18.7	259	72.1	5.7	281	20.4
mac_econ_fwd500	6.5	344	15.7	5.1	268	18.9	7.0	329	21.3
pwtk	12.0	311	31.5	11.4	277	41.3	7.5	289	25.8
Ga41As41H72	11.9	338	35.1	9.7	275	35.1	5.9	290	20.4
F1	10.1	330	30.5	8.1	275	29.3	8.6	298	28.9
af_0_k101	15.4	330	46.7	13.5	272	49.5	7.5	278	27.0
af_shell9	14.6	330	44.3	13.0	275	47.1	22.6	300	75.2
kron_g500-logn19	3.2	339	9.6	2.9	257	11.1	7.0	291	24.0
mc2depi	16.1	333	31.4	10.9	268	40.5	6.0	323	18.6
dielFilterV2clx	9.6	348	27.7	7.1	279	25.3	5.4	308	17.6
Stanford_Berkeley	6.2	297	20.9	6.0	255	23.5	4.9	292	16.6
web-BerkStan	7.3	316	23.1	6.2	256	24.1	4.8	283	17.1
flickr	4.0	298	13.5	3.5	257	13.5	3.2	288	10.9
12month1	7.0	284	24.7	5.5	243	22.7	5.8	289	20.1
webbase-1M	4.8	296	14.0	4.6	251	18.4	4.0	334	11.9
rail4284	8.1	296	21.4	6.2	246	25.0	7.2	320	22.6
dielFilterV3real	10.6	340	31.1	8.4	278	30.2	6.0	315	19.1
dielFilterV2real	8.0	339	23.6	6.4	282	22.7	8.9	340	26.1
atmosmodd	25.2	329	76.5	18.4	268	68.6	3.8	307	12.3
StocF-1465	10.4	351	29.6	7.5	282	26.5	4.5	309	14.5
as-Skitter	4.3	298	14.3	3.7	267	13.9	2.6	280	9.2
kkt_power	4.3	315	13.5	3.6	277	13.0	3.1	281	11.1
delaunay_n21	8.7	321	27.2	6.4	263	24.1	2.7	292	9.2
circuit5M_dc	7.8	338	23.0	6.5	258	25.3	2.5	290	8.5
nlpkkt120	9.5	340	28.1	18.2	271	67.0	4.6	281	16.4
channel-500x100x100-b050	5.8	329	17.6	13.9	268	51.7	7.3	340	21.5
hugetric-00000	6.1	281	21.7	4.8	250	19.2	1.5	233	6.4
hugetric-00010	3.6	290	12.4	3.2	258	12.5	1.2	220	5.2
italy_osm	8.1	291	27.9	6.2	261	23.8	1.8	219	8.2
adaptive	10.8	318	34.1	8.2	247	33.1	1.9	254	7.6
hugetric-00020	3.6	278	12.9	3.3	261	12.5	1.1	260	4.2
great-britain_osm	7.2	314	22.8	5.7	259	21.9	1.6	254	6.3
delaunay_n23	7.9	333	23.7	6.1	263	23.3	2.3	281	8.2
germany_osm	6.2	307	20.2	4.8	258	18.5	1.5	251	6.0
asia_osm	7.8	313	24.8	5.7	254	22.6	1.7	256	6.7
hugetrace-00010	4.6	280	16.5	3.9	260	15.0	1.2	234	5.2
road_central	2.6	324	8.1	2.3	246	9.2	0.8	268	3.0
hugetrace-00020	3.6	282	12.6	3.5	242	14.3	1.0	220	4.7
hugebubbles-00000	5.4	287	18.7	4.5	262	17.0	1.2	244	5.0
hugebubbles-00010	3.5	279	12.4	3.5	247	14.0	0.9	229	4.0
hugebubbles-00020	3.5	278	12.4	3.6	244	14.5	0.9	230	4.1
road_usa	4.2	304	13.8	3.5	255	13.6	1.7	301	5.7

Fig. 8. Measured performance and power for all reported sparse matrices.

[12] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, ser. SPAA'09. New York, NY, USA: ACM, 2009, pp. 233–244.

[13] A. Buluc, S. Williams, L. Oliker, and J. Demmel, "Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS'11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 721–733.

[14] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 451–460. [Online]. Available: http://doi.acm.org/10.1145/1815961.1816021

[15] "Joulemeter: Computational Energy Measurement and Optimization." [Online]. Available: http://research.microsoft.com/en-us/projects/joulemeter/

[16] I. Park and R. Buch, "Improve debugging and performance tuning with etw," 2007. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc163437.aspx