Sparse Matrix-Vector Multiply on the Keystone II Digital Signal Processor

Yang Gao[1,2], Fan Zhang[1,2], and Jason D. Bakos[1,3]

[1]Department of Computer Science and Engineering, University of South Carolina, Columbia, SC, USA
[2]{gao36,zhangf}@email.sc.edu
[3]jbakos@cse.sc.edu

***Abstract*—In this paper we describe an implementation of sparse matrix-vector multiply (SpMV) on the Texas Instruments (TI) Keystone II architecture. The Keystone II is an eight core Digital Signal Processor (DSP) that offers floating point performance comparable to a desktop CPU while having a power envelope comparable to a mobile embedded CPU. This, combined with its integrated communication interfaces, potentially make it a scalable and efficient HPC processor technology. For this architecture, the key to achieving high computational efficiency is the careful use of its on-chip scratchpad memory. SpMV is a HPC kernel that is both memory bounded and has an irregular memory access pattern. When tuning this kernel, we found that using scratchpad can provide as much as 50% improvement in effective memory bandwidth as compared to using cache, but only with careful scratchpad allocation and run-time management. This includes selection of tile size, the mapping of arrays to specific on-chip memory structures, and the methods by which the DMA is performed in parallel with computation.**

I. Introduction

It has recently become common practice to integrate co-processors into high-performance computers, and graphical processor units (GPUs) are currently the most common co-processor technology. GPU coprocessors generally improve overall energy efficiency as compared with systems having only CPUs [1][2]. Despite this, energy efficiency remains a major constraint in high performance computer design: the most efficient HPC systems are limited to 4.5 Gflops/W, implying that an exascale machine would require 100s of MW to operate at full capacity using current technology. This has served as a motivation to explore radically new processor technologies. One potential alternative to traditional CPU+GPU processors are Digital Signal Processors (DSPs), whose principal difference from CPU and GPU architectures is that they are statically scheduled and rely mainly on program and compiler optimization for performance. Of the newest generation of DSP, the Texas Instruments Keystone II architecture is also unique in that it adopts a system-on-chip (SoC) design, similar to embedded processors, integrating ARM-based CPUs, DSP-based coprocessors, and high speed network interfaces onto a single die. When clocked at 1.35 GHz, an eight-core C66 device has a peak theoretical throughput of 172.8 single precision Gflops and achieves 80 sustained Gflops/s for single precision general matrix-matrix multiply (SGEMM) using the current version of TI's BLAS library[3]. The C66 lacks out-of-order and speculative execution. Instead, it exploits instruction level parallelism using an eight-way very long instruction word (VLIW). C66 cores are loosely coupled and thus do not include a shared last-level cache. Their onchip memory can be reconfigured by the software such that a portion or all of one or both level of caches can be used as a scratchpad memory. There is also a separate scratchpad memory that are shared among the cores. This flexibility provides the ability to improve memory system performance by mapping different data structures to either cache, scratchpad, or both. In this study, we chose the sparse matrix-vector multiply kernel using the Compressed Sparse Row (CSR) format as a case study. This kernel's performance is closely correlated with the achieved performance of the memory hierarchy due to its low arithmetic intensity and high memory intensiveness.

II. SpMV Kernel

SpMV performs the computation $Y = A\alpha X + \beta Y$, where A is a matrix stored in a sparse format, X and Y are vectors stored as dense 1D arrays, and α and β are scalars. Our SpMV kernel uses the popular Compressed Sparse Row (CSR) sparse matrix format, where matrix A is represented using three one-dimensional arrays, **val**, **col**, and **ptr**. The **val** array holds each of the matrixs non-zero values in ascending column and row order, while the **col** array holds each values corresponding column index. The **ptr** array is indexed by row and holds the position within the **val** and **col** array where each matrix row begins. For example, an M x N matrix where M = 2 could be stored using arrays: $val = \{2,4,6,8,10,12\}$, $col = \{2,3,4,5,3,5\}$, and $ptr = \{0,4,6\}$. In this case, the matrix contains $ptr[M] = 6$ nonzero elements, the second row contains $ptr[2] - ptr[1] = 2$ elements, and the second element of row 1 is $val[ptr[1]+1] = 12$ in column $col[ptr[1]+1] = 5$. There are several reasons why sparse matrix-vector multiply with CSR format is a notoriously difficult kernel for which to achieve high functional unit utilization. First, the **col** array imposes gather-style indirect references to the input vector X, and the locality of the irregular accesses to X depends on the distribution of populated columns (defined in the col array). Second, the unpredictable number of entries per matrix row, as defined by the **ptr** array, requires dynamic control behavior when computing the reduction operation when accumulating

978-1-4799-6233-4/14/$31.00 ©2014 IEEE

the inner product. Third, the entire operation is generally memory-bound for modern processors, requiring roughly 3/8 floating point operations per byte for single precision values and 32-bit indices, where n is average number of entries per matrix row (shown later in Equation 2Efficiencyequation.4.2). As a result of these challenges, modern state-of-the-art CPUs and GPUs generally achieve 1-5% of their peak throughput for this computation depending on the density and structure of the matrix [4].

III. Implementation

As is the case with GPUs, the C66 achieves high performance only when the software is carefully hand-tuned to exploit specific aspects of its architecture. Unfortunately, unlike GPUs, there is little established methodology or best practices for kernel tuning. In this section we describe how we map the SpMV naïve implementation to a DMA supported double buffer kernel with optimized usage of the scratchpad memory.

A. Naïve implementation

The basic implementation of SpMV was a simple, naïve loop that directly performs the kernel as shown in Algorithm 1Naïve Implementationalgorithm.1.

Algorithm 1 Naïve Implementation

Input: $val, col, ptr, y, x, \alpha, \beta$

1: $row \leftarrow initial\_row$
2: **for** $i = coreNum \times (M/cores) \rightarrow (coreNum + 1) \times (M/cores) - 1$ **do**
3: **if** $ptr\,[row] == i$ **then**
4: $row \leftarrow row + 1$
5: $y\,[row] \leftarrow y\,[row] \times \beta$
6: **end if**
7: $y\,[row] \leftarrow y\,[row] + \alpha \times val\,[i] \times x\,[col\,[i]]$
8: **end for**

B. Mapping to DSP Double Buffered implementation

Each DSP core has a 32KB L1 SRAM and 1024KB L2 SRAM that can be programmatically configured to behave as a traditional cache, as a program-controlled scratchpad memory, or as a combination of both, with arbitrary portions assigned as cache and scratchpad. This organization resembles the shared memory in GPUs, allowing shared access within a thread block. Besides the L1 and L2 on-chip memory, another 6MB SRAM called the Multicore Shared Memory Controller (MSMC) is also available to all eight DSP cores. Since this memory is sharable by all cores, there is no equivalent on contemporary GPUs. To support the on-chip memories, the device also contains an integrated direct data access (DMA) controller that allows data to be exchanged between on- and off-chip memories (and between on-chip memories) in parallel to operations being performed on the DSP. The DMA controller can also be programmed to perform complex 3-dimensional data access patterns on both the source and destination memories. Data accessed in a predictable, regular access pattern can be concurrently prefetched using EDMA to exchange data between scratchpad and DRAM. Data accessed irregularly in a data-dependent pattern can be cached in a separate region of scratchpad to take advantage of locality. In our implementation, we allocated a portion of the L2 cache as a scratchpad and used the DMA controller to implement a double buffer for the **val** and **col** arrays. The input vector, output vector, and **ptr** could be simply cached or mapped to a circular buffer.

Our SpMV implementation is also tuned to maximize functional unit utilization. We used loop fission to separate the control independent calculations from the dependent calculations: a "product loop" and an "accumulate loop" as illustrated in Algorithm 2Loop Fissionalgorithm.2. Each loop is tiled, processing each block of M entries allocated to the scratchpad SRAM. In order to maximize the utilization of the 8-way VLIW instructions, the TI compiler attempts to software pipeline any loop that doesn't contain branch instructions or function calls. Software pipelining allows the compiler to break dependencies within the loop body and improve the functional unit utilization at the cost of increased register usage (each core has two 32 x 32 bit register files). The product loop has no dependencies and can be software pipelined by the compiler, resulting in high computational performance, but is limited by memory bandwidth to store the products. In order to further speed up the product loop we allocated L1 scratchpad memory to hold the product array. The IF statement on line 3 of Algorithm 1Naïve Implementationalgorithm.1 prevents the compiler from applying software pipelining. In order to enable this feature, we converted the code to assembly language and implemented the conditional code with predicated instructions as opposed to a branch instruction as was used by the compiler-generated code. The TI assembler was able to software-pipeline this assembly language.

We used DMA to transfer blocks of the **val** and **col** arrays to L2 scratchpad. This allowed for the DSP core to process the previous block while transferring the next block from DRAM. However, the if-statement in Algorithm 1Naïve Implementationalgorithm.1 and Algorithm 2Loop Fissionalgorithm.2 makes it difficult to apply this technique to the **Y** and **ptr** arrays. Although these arrays are accessed as a contiguous block of data, they do not need to be transferred at the same time as **val** and **col** because they are consumed at a different rate (determined by matrix sparsity).

Specifically, assuming all scratchpad buffers are of equal size, a new **Y** and **ptr** buffer would need to be transferred from memory n/r times less frequently than the **val** and **col** buffers, where n = size of the **val** array and r = size the of **ptr** array. As such, in order to add DMA support to the **Y** and **ptr** buffer we must synchronize all DMA transfers. In order to do this, we must be able to initiate a new DMA transfer to fill a **Y** and **ptr** scratchpad buffer before it has been completely emptied. Internal DMA (IDMA) is a feature that is designed to transfer content between the on-chip memories. Using this feature we implemented a special circular buffer scheme for **ptr** and **Y**

Algorithm 2 Loop Fission

Input: $val, col, ptr, y, x, \alpha, \beta$

```
1: for i = 0 → M do //product loop
2:     prod[i] ← α × val[i] × x[col[i]]
3: end for
4: Acc ← 0
5: for i = 0 → M step by K do //accumulation loop
6:     Acc ← Acc + prod[i]
7:     if ptr[row] == i then
8:         row ← row + 1
9:         y[row] ← y[row] × β + Acc
10:        Acc ← 0
11:    end if
12:    Acc ← Acc + prod[i + 1]
13:    if ptr[row] == i + 1 then
14:        row ← row + 1
15:        y[row] ← y[row] × β + Acc
16:        Acc ← 0
17:    end if
18: ...
19:    Acc ← Acc + prod[i + K]
20:    if ptr[row] == i + K then
21:        row ← row + 1
22:        y[row] ← y[row] × β + Acc
23:        Acc ← 0
24:    end if
25: end for
```

which is depicted in Figure 1(a) the logical organization of the buffer, (b) buffer state before the accumulation loop, (c) buffer state after the accumulation loop, (d) IDMA and DMA transfers performed.figure.caption.1.

Assume we have one scratchpad buffer for the **ptr** array with c entries and a traditional double buffer for the **val** array named $\mathbf{val_A}$ and $\mathbf{val_B}$. When the kernel consumes the last value in $\mathbf{val_A}$ assume that n remain unread in the **ptr** buffer. At this time, the kernel actives the full $\mathbf{val_B}$ buffer and initiates a new DMA transfer to fill $\mathbf{val_A}$ from DRAM. If the **ptr** buffer is less than half full, the kernel uses IDMA to copy the remaining entries to the beginning of the **ptr** buffer, and then (after a barrier) initiates a DMA from DRAM to replenish the buffer. Assuming the matrix has, on average, at least one entry per row, the **ptr** array will not be consumed faster than the **val** array and this method will function correctly.

Note that for most matrices the IDMA copy will not be performed every time a **val** buffer is emptied. When the IDMA transfer does occur, it must complete before the subsequent DMA transfer from DRAM. However, instead of waiting, the kernel overlaps the IDMA transfer with the product loop in Algorithm 2Loop Fissionalgorithm.2.

The accumulation loop in Algorithm 2Loop Fissionalgorithm.2 checks if value i is the first value of its row before every add operation (line 21). This adds a substantial amount of overhead to this loop. In order to reduce

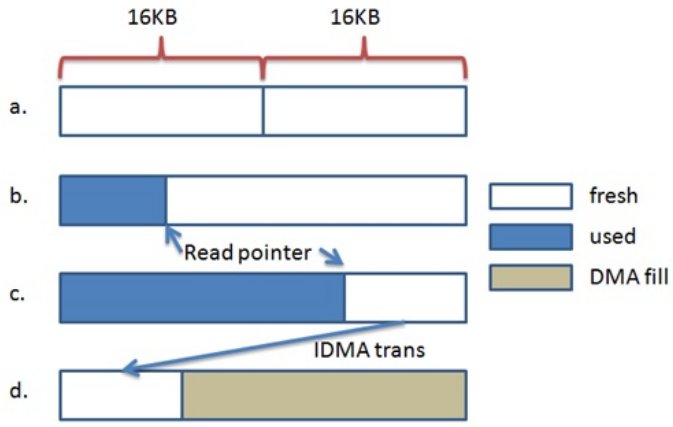

Fig. 1: (a) the logical organization of the buffer, (b) buffer state before the accumulation loop, (c) buffer state after the accumulation loop, (d) IDMA and DMA transfers performed.

this overhead, we added an optimization where, at the start of the loop body, the code checks the number of values remaining on the current row, i.e. $ptr[row+1]ptr[row]$. If this value is > 8, a new inner loop can perform $(ptr[row+1]ptr[row])/8$ unrolled iterations without checking the row pointer. We arrived at the unroll factor of 8 by tuning. In addition, we also use the 2-way SIMD ADD to further improve the performance. This optimization has more of a benefit for denser matrices (Table IIISpMV Performance on DSP, CPU, and GPUtable.caption.5).

C. Tile Size and Buffer Locations

As shown in Figure 2Memory System Usagefigure.caption.2, on chip buffers can be allocated in L1 scratchpad, L2 scratchpad, shared memory (MSMC) or referenced from DRAM and cached. The L1 SRAM has the lowest latency but has a small capacity (32KB), which makes it difficult to perform large enough DMA transfers to offset the start-up overheads required to initiate a DMA transfer. The L2 memory is 32 times larger but has less bandwidth and higher latency relative to the DSP. The MSMC is tightly integrated with the DRAM controller in the Keystone II design, but it lacks support for cache coherency. Using cache avoids the DMA start-up overhead but may be subject to a high conflict miss rate.

These features provides the programmer flexibility but allocating and managing the on-chip memory requires decisions that are governed by a complex set of trade-offs. One of the important trade-offs to consider when using program controlled scratchpad is that the programmer must add additional branch instructions to the code in order to maintain the buffers. The number of additional branch instructions can be reduced when multiple buffers that have equal consumption rate can be set to the same size and thus their management can be synchronized. However, allocation of these buffers is constrained by the access pattern of the corresponding data structure.

TABLE I: Buffer Location and Performance

Non-zeroes per row	val	col	ptr	y	prod	Gflops	Norm. Perf.[1]	Note
3	S	L2	L2	L2	L1	2.26	1.57	Best
	S	L2	L2	L2	L2	1.84	1.28	Median
	L2	L2	C	C	S	1.23	0.85	Worst[2]
	C	C	C	C	C	1.44	1	All Cache
151	L2	S	L2	L2	L2	3.76	1.50	Best
	S	C	L2	L2	S	3.55	1.41	Median
	C	C	C	C	L2	2.66	1.06	Worst[2]
	C	C	C	C	C	2.51	1	All Cache

L1:level 1 SPRAM, L2:level 2 SPRAM, S:MSMC, C:cache
1: The results are normalized to the all cache configuration
2: The worst amongst the configurations with SPRAM

The **val** and **col** buffers are most flexible and could be placed all locations and simply cached. The circular buffer of **ptr** and **Y** rely on IDMA to achieve both efficiency and overlapping with computation. These buffers cannot be allocated to MSMC because it cannot be written with internal DMA. The **prod** buffer is an internal data array that is accessed sequentially.

Our SpMV kernel is parameterizable, allowing each buffer to be sized and allocated at run-time using a given configuration. In order to evaluate the impact of allocation decisions, we ran the kernel against a full enumeration of valid buffer mappings. In order to test matrices with difference sparsity, we used two test matrices: one containing 3 consecutive elements per row and one with 151 elements per row.

Table IBuffer Location and Performancetable.caption.3, lists the best, median, and worst performance given by set of scratchpad mappings, as well the performance given when using cache for all arrays. Note that both test matrices give a different best result, meaning that best tile size and mapping configuration depends on the density of the input matrix. The performance is most sensitive to the usage of L1 and the tile size. The largest possible tile size is 16KB when mapping the prod array to L1. This configurable gives us the best result when processing the sparser matrix.

However, when processing the denser matrix it takes longer for the kernel to reach the end of the row. This reduces the average number of instructions per matrix value, allowing the buffers to be consumed faster and shifts the bottleneck to the background DMA transfers. As a result, in this case a larger buffer is more favorable.

Table IBuffer Location and Performancetable.caption.3 also lists the configurations that emphasizes the impact of using the cache instead of the scratchpad. The results are normalized to the pure cache mapping which serves as the baseline. We found that the pure scratchpad implementation achieves about 50% speedup as compared with using cache.

For the results shown below, we use the best allocation given by the sparser matrix, since our test matrices are closer in density to 3 elements/row than 151 elements/row [4] [5].

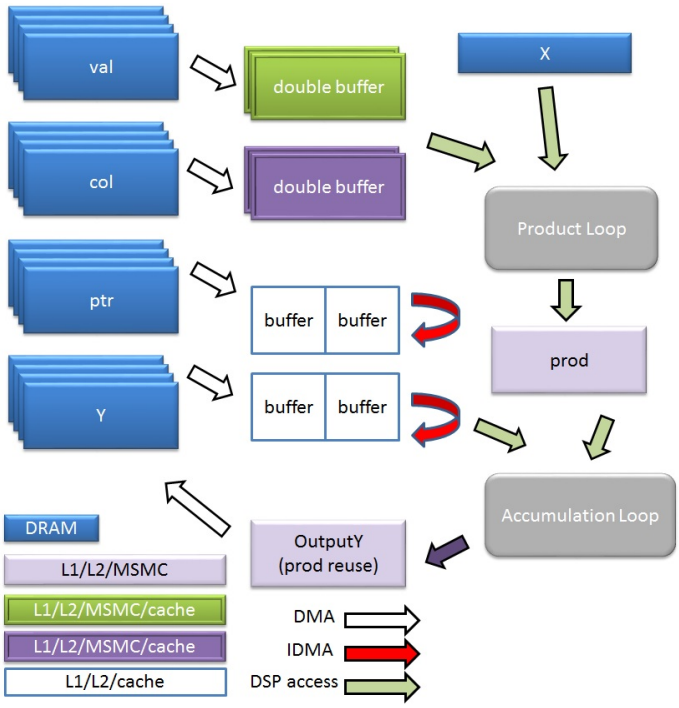

Fig. 2: Memory System Usage

IV. Efficiency

Recall that SpMV performs the operation $Y = A\alpha X + \beta Y$. For single precision values and 32-bit indices, each non zero value in A, requires two multiplies ($A_{ij} \times \alpha \times X_j$) and one add (to compute the dot product). This requires a load of a four-byte A_{ij} value, a four-byte column index j, and a four-byte vector value X. If we assume that X is transferred only once from RAM, these three operations require $(8+4)/rows$ bytes due to compulsory misses.

For each matrix row, the kernel performs an additional multiply ($\beta \times Y_i$) and add (to Y_i) which requires a four-byte load and subsequent store to Y and a four-byte load from the ptr array. As such, SpMV has an arithmetic intensity (AI) of

$$AI = \frac{nnz \times 3ops + rows \times 2ops}{nnz \times 2 \times 4bytes + rows \times 3 \times 4bytes + cols \times 4bytes} \quad (1)$$

$$= \frac{nnz \times 3 + rows \times 2}{nnz \times 8 + rows \times 12 + cols \times 4} ops/byte \quad (2)$$

For most modern architectures, this level of arithmetic intensity makes SpMV a memory-bound operation. This allows us to compute the performance bound as the product of arithmetic intensity and peak memory bandwidth. Our platform has a peak memory bandwidth of 12.8 GB/s on one DRAM interface, giving a performance bound of $12.8 \times AI$, while our desktop GPU has a peak memory bandwidth of 192.3 GB/s, giving a throughput bound of $192.3 \times AI$ Gflops.

The ratio of actual performance to the performance bound is the efficiency, defining how much of the available memory

TABLE II: Summary of Test Platforms

	Intel i7 3770K MKL	NVIDIA GTX 680 cuSPARSE	NVIDIA Tegra K1 cuSPARSE	TI 6638K2K
Arch	Ivy Bridge	Kepler	Kepler	KeystoneII
Memory B/W	25.6 GB/s	192.3 GB/s	17.1 GB/s	12.8 GB/s
TDP	77 W	195 W	∼10 W	∼15 W
SPRAM KB/core	n/a	64/64[1]	64/64[1]	32/1024/768[2]
Single Precision Peak Throughput	448 Gflops	3090 Gflops	365 Gflops	@1.35 GHz 172.8 Gflops (DSP) +44.8 Gflops (ARM)

1: Register file/allocable share memory or L1.
2: L1 SRAM / L2 SRAM / MSMC per core

bandwidth is actually being used by the kernel. This metric is an important indicator of the effectiveness of the kernel implementation and the memory system (e.g. cache).

V. Test Platforms

As shown in Table IISummary of Test Platformstable.caption.4, our reference CPU is a four-core Core i7 CPU and our test GPUs are a desktop GPU (NVIDIA GTX 680) and embedded GPU (Tegra K1). Our CPU results are given by Intel's Math Kernel Library (MKL) [6] and our GPU results are given by the NVIDIA cuSPARSE [7] library. Since these libraries were developed and optimized by Intel and NVIDIA, respectively, we assume that their performance represents a reasonable evaluation of the maximum capabilities of the corresponding processors.

VI. Experimental Results

Table IIISpMV Performance on DSP, CPU, and GPUtable.caption.5 lists the performance results of the DSP, CPU, and GPU using matrices from and the Matrix Market [4] and University of Florida Matrix Collection [5]. The table is sorted by matrix density, in terms of average number of elements per row, and shows the raw performance, power efficiency, and percentage of memory bandwidth used (shown as kernel efficiency) for each matrix.

In our experiment, the CPU generally achieves higher memory efficiency than the DSP. We assume this is because Intels memory system is able to pre-fetch the cache blocks for each of the sequentially-accessed arrays (**val**, **col**, **ptr**, and **Y**) before they generate cache misses, which may approximate the behavior of double buffering using DMA. Also, because of its larger caches, it may achieve higher performance when accessing the **X** array.

One of the parameters in Equation 1Efficiencyequation.4.1 is number of matrix entries per row, which we use to calculate the access rate of the **Y** and **ptr** arrays. For the test cases below, we specify this as an average, since none of matrices contain a consistent number of entries per row. In one matrix (lhr71c), a high variance in the number of entries per row causes our calculated arithmetic intensity to appear lower than it is in reality. This matrix also has a relatively small **X** vector, yielding a high achieved efficiency, leading to a calculated efficiency of $> 100\%$.

As compared to the desktop GPU, the DSP generally achieves 5 - 10 X less performance while having 15 X less memory bandwidth. This is because the DSP is generally able to achieve twice the memory efficiency of the GPU. The DSP consistently outperforms the embedded GPU, despite the embedded GPU's higher peak memory bandwidth (17.1 GB/s vs. 12.8 GB/s). The DSP's memory efficiency is comparable to the CPU for the denser matrices, despite the CPU having a TDP that is 5 X that of the DSP.

VII. Related Work

Optimizing sparse matrix-vector multiply on emerging architectures has been the subject of much recent work [8–11] . To our best knowledge, this is the first implementation of SpMV on the current generation of the TI Keystone II architecture. Our previous work described a similar SpMV implementation on TIs Keystone architecture[12].

There has been some recent work in dense linear algebra on this architecture, demonstrating 80 single precision Gflops for SGEMM[13]. There has also been recent interest in improving the performance or power efficiency of SpMV on GPUs [14–20].

VIII. Conclusion

In this paper we described the optimization and performance of a sparse matrix vector multiply kernel on the TI Keystone II architecture. We were able to achieve a 50% performance improvement as compared to using cache by using allocation and runtime management of the onchip scratchpad memory.

On average for our test matrices, our performance results show a 2.7X and 7.4X slowdown compared to the CPU and desktop GPU respectively, and a 1.27 speedup compared to the mobile GPU. The DSP's low performance as compared with the CPU and desktop GPU can be attributed to the DSPs relatively low memory bandwidth. Despite this, the DSP is able to utilize roughly 2 X of its peak memory bandwidth as compared to the desktop and embedded GPUs.

IX. Acknowledgements

We would like to thank Arnon Friedmann, Murtaza Ali, and Alan Ward from Texas Instruments and Emily Teng from Advantech Corporation for their support of this work. This material is based upon work supported by Texas Instruments and the National Science Foundation under grant No. 0844951.

References

[1] J. Enos, C. Steffen, J. Fullop, M. Showerman, G. Shi, K. Esler, V. Kindratenko, J. E. Stone, and J. C. Phillips, "Quantifying the impact of gpus on performance and energy efficiency in hpc clusters," in *Green Computing Conference, 2010 International*. IEEE, 2010, pp. 317–324.

[2] S. Hemmert, "Green hpc: From nice to necessity," *Computing in Science and Engineering*, vol. 12, no. 6, pp. 8–10, 2010.

TABLE III: SpMV Performance on DSP, CPU, and GPU

Matrix Name	Ave. entries/row	Performance (Gflops)				Kernel Efficiency			
		DSP	CPU	GPU1[1]	GPU2[2]	DSP	CPU	GPU1	GPU2
mc2depi	4.0	2.16	5.46	17.79	1.64	0.58	0.73	0.32	0.33
shyy161	4.3	1.56	5.71	15.43	1.55	0.41	0.75	0.27	0.31
scircuit	5.6	2.02	4.89	14.30	1.35	0.51	0.62	0.24	0.26
ASIC_100ks	5.8	2.21	5.55	11.01	1.43	0.55	0.70	0.18	0.27
mac_econ_fwd500	6.1	2.13	6.50	12.51	1.36	0.53	0.81	0.21	0.25
thermal1	6.9	2.09	4.32	11.19	1.14	0.56	0.58	0.20	0.23
lhr71c	21.7	2.34	9.86	16.54	1.89	0.52	1.08	0.24	0.31
ldoor	24.9	2.66	6.78	22.82	2.36	0.58	0.74	0.33	0.39
shipsec1	25.0	2.98	7.73	23.63	2.48	0.65	0.84	0.34	0.41
pwtk	52.9	2.27	7.63	24.70	2.37	0.50	0.83	0.36	0.39
cant	64.2	3.08	8.79	23.97	2.54	0.67	0.95	0.35	0.41
consph	72.1	3.30	8.16	22.98	2.58	0.71	0.88	0.33	0.42
audikw_1	82.3	3.21	7.53	21.79	2.63	0.69	0.81	0.31	0.43
m_t1	99.9	3.44	7.94	21.58	2.63	0.73	0.85	0.31	0.42
pdb1HYS	119.3	3.73	8.82	20.85	2.60	0.79	0.94	0.30	0.42
TSOPF_FS_b300_c3	155.6	2.96	6.60	30.30	2.57	0.63	0.70	0.43	0.41
AVERAGE	47	2.63	7.02	19.5	2.07	0.60	0.80	0.29	0.35

1: GTX 680 2: Tegra K1

[3] M. Ali, E. Stotzer, F. D. Igual, and R. A. van de Geijn, "Level-3 blas on the ti c6678 multi-core dsp," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*. IEEE, 2012, pp. 179–186.

[4] R. F. Boisvert, R. Pozo, K. A. Remington, R. F. Barrett, and J. Dongarra, "Matrix market: a web resource for test matrix collections." in *Quality of Numerical Software*, 1996, pp. 125–137.

[5] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[6] Intel, "Intel math kernel library." [Online]. Available: https://software.intel.com/en-us/intel-mkl/

[7] NVIDIA, "Cublas libraries." [Online]. Available: https://developer.nvidia.com/cusparse

[8] Y. Shan, T. Wu, Y. Wang, B. Wang, Z. Wang, N. Xu, and H. Yang, "Fpga and gpu implementation of large scale spmv," in *Application Specific Processors (SASP), 2010 IEEE 8th Symposium on*. IEEE, 2010, pp. 64–70.

[9] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix–vector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178–194, 2009.

[10] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Performance evaluation of the sparse matrix-vector multiplication on modern architectures," *The Journal of Supercomputing*, vol. 50, no. 1, pp. 36–77, 2009.

[11] 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*. ACM, 2009, p. 18.

[12] Y. Gao and J. D. Bakos, "Sparse matrix-vector multiply on the texas instruments c6678 digital signal processor," in *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*. IEEE, 2013, pp. 168–174.

[13] F. D. Igual, M. Ali, A. Friedmann, E. Stotzer, T. Wentz, and R. van de Geijn, "Unleashing dsps for general-purpose hpc," 2012.

[14] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on gpus," in *ACM Sigplan Notices*, vol. 45, no. 5. ACM, 2010, pp. 115–126.

[15] F. Vazquez, G. Ortega, J.-J. Fernández, and E. M. Garzon, "Improving the performance of the sparse matrix vector product with gpus," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*. IEEE, 2010, pp. 1146–1151.

[16] H. Anzt, M. Castillo, J. C. Fernández, V. Heuveline, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí, "Optimization of power consumption in the iterative solution of sparse linear systems on graphics processors," *Computer Science-Research and Development*, vol. 27, no. 4, pp. 299–307, 2012.

[17] H. Anzt, V. Heuveline, J. I. Aliaga, M. Castillo, J. C. Fernández, R. Mayo, and E. S. Quintana-Ortí, "Analysis and optimization of power consumption in the iterative solution of sparse linear systems on multi-core and many-core platforms," in *Green Computing Conference and Workshops (IGCC), 2011 International*. IEEE, 2011, pp. 1–6.

[18] A. J. Wijs and D. Bošnački, "Improving gpu sparse matrix-vector multiplication for probabilistic model checking," in *Model Checking Software*. Springer, 2012, pp. 98–116.

[19] J. Godwin, J. Holewinski, and P. Sadayappan, "High-performance sparse matrix-vector multiplication on gpus for structured grid computations," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*. ACM, 2012, pp. 47–56.

[20] W. Xu, H. Zhang, S. Jiao, D. Wang, F. Song, and Z. Liu, "Optimizing sparse matrix vector multiplication using cache blocking method on fermi gpu," in *Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing (SNPD), 2012 13th ACIS International Conference on*. IEEE, 2012, pp. 231–235.