Sparse Matrix-Vector Multiply on the Texas Instruments C6678 Digital Signal Processor

Yang Gao
Department of Computer Science and Engineering
University of South Carolina
Columbia, SC 29208, USA
Email: gao36@email.sc.edu

Jason D. Bakos
Department of Computer Science and Engineering
University of South Carolina
Columbia, SC 29208, USA
Email: jbakos@cse.sc.edu

***Abstract*—The Texas Instruments (TI) C6678 "Shannon" is TI's most recently-released Digital Signal Processor (DSP). Although its original purpose was voice and video encoding and decoding, it may have the potential to become a practical coprocessor for scientific computing. In this paper, we evaluate the C6678 in terms of its programming methodology, performance, and power efficiency. As a case study, we implemented a sparse matrix vector multiply (SpMV) kernel and used it to perform a comparative study against the NVIDIA Kepler GK104 and GK106 Graphical Processor Units. On the DSP, we take advantage of many of the C6678's features, including its VLIW and SIMD instruction set architecture, program-controlled scratchpad memory, and direct memory access (DMA) controller. We found that the DSP is unable to outperform the GPUs in raw performance but can achieve roughly equal power efficiency in Gflops/Watt. This is more impressive when considering that the DSP is manufactured in a 45 nm process while the GPUs are manufactured in a 28 nm process. We believe that subsequent DSPs, when manufacturered in a modern fabrication process, may be more competitive with GPUs in power efficiency. We also found that, for this kernel, the DSP is able to achieve higher utilization of both its peak memory bandwidth and its functional units as compared with the GPUs. In this paper we describe our kernel and the programming techniques required to optimize its performance.**

***Keywords*—high performance computing; sparse matrix vector (SpMV); linear algebra; digital signal processor (DSP); graphical processor unit (GPU); very long instruction word (VLIW)**

I. Introduction

It has recently become common practice to integrate coprocessors into high-performance computers, and graphical processor units (GPUs) are currently the most common coprocessor technology. High performance computers with GPU coprocessors generally achieve higher overall power efficiency as compared with those with only CPUs. An experiment performed at the National Center for Supercomputing Applications evaluated a cluster comprised of 64 dual-core Opteron 2216 processors and 128 NVIDIA S1070 GPUs for four scientific benchmark applications and observed a 3 to 23 times improvement in power efficiency as compared to using CPUs alone [1]. GPU-accelerated computers have also recently become the second, third, and fourth most power-efficient computers on the "Green 500" list [2]. While this sounds encouraging, the most efficient systems on this list are still limited to an efficiency of 2.5 Gflops/W, implying that an exascale machine would require 100s of MW to operate at full capacity using current technology.

The Texas Instruments CorePac architecture may have the potential to achieve higher power efficiency than GPUs for scientific kernels. PCI-express-based add-in cards containing four and eight C6678 DSPs are already available, allowing these DSPs to be used in the same heterogeneous computing model as GPUs. However, unlike GPUs, these DSPs have integrated network interfaces and are capable of running an operating system. Thus in theory they could participate in a distributed processing system with little or no involvement from CPU-based hosts. This may lead to next generation heterogeneous systems that have less emphasis on CPUs and achieve a revolutionary improvement in overall power efficiency.

Much like how the GPU's role as a coprocessor was an outgrowth of its initial market in the 3D gaming industry, the TI DSP is already widely manufactured and deployed in many of the world's cellular phone base stations. Much like the GPU, its primary market will continue to sustain its continued development while it grows in a potential secondary role as a coprocessor for supercomputing.

When clocked at 1 GHz, the C6678 has a peak theoretical throughput of 128 single precision Gflops (16 operations per clock per core) and achieves 80 sustained Gflops for single precision general matrix-matrix multiply (SGEMM) using the current version of TI's BLAS library [3]. It has 12.8 GB/s of DDR3 DRAM bandwidth.

The C6678's design has several features that could make it highly power-efficient when used with carefully optimized and parallelized code. It lacks power-hungry features such as out-of-order and speculative execution and instead exploits instruction level parallelism using an eight-way very long instruction word (VLIW). Its eight on-chip cores are loosely-coupled, as they do not include coherent mid-level caches nor a shared last-level cache. Instead it relies on explicit inter-core communication to exploit core-level parallelism. Each core has two levels of cache, but the caches can be reconfigured by the software such that a portion or all of one or both level of caches can be used as a software controlled scratchpad memory. There is also a separate, shared software-controlled scratchpad memory. The C6678 supports predicated instructions in order to prevent small sections of conditional code from degrading instruction throughput or VLIW utilization.

In this study we targeted the sparse matrix-vector multiply

kernel using the Compressed Sparse Row (CSR) format as a case study. We chose this particular kernel for several reasons. First, SpMV is an important kernel for many scientific applications. Second, GPU, SIMD, and vector processors generally achieve a low computational and memory efficiency for this kernel due to its irregular computational and memory pattern and its low arithmetic intensity. Third, SpMV is representative of a large class of kernels that operate on sparse data structures, such as (such as GRAPH500[4] and the Fast Multipole Method [5]).

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 matrix's non-zero values in ascending column and row order, while the **col** array holds each value's 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 $M \times N$ 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 performance. First, the **col** array imposes gather-style indirect references to the input vector **X**, and the locality of the irregular accesses to **X** depends 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 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 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 [6]. The most difficult matrices to multiply are those that are very sparse.

III. Implementation

As is the case with GPUs, the C6678 achieves high performance only when the software is carefully hand-tuned to its architecture. In this section we describe the steps taken to tune the SpMV kernel in order to establish a general methodology for DSP code optimization. Note all the performance results in this section are based on a tri-diagonal matrix input with 10 million rows.

A. Initial implementation

Our initial implementation of SpMV was a simple, naïve loop that directly performs the kernel as shown in Algorithm 1.

Algorithm 1 Naïve Implementation

```
row ← initial_row
for i = corenum × (M/cores) → (corenum + 1) × (M/cores) − 1 do
    if ptr[row] == i then
        row ← row + 1
        y[row] ← y[row] × β
    end if
    y[row] ← y[row] + α × val[i] × x[col[i]]
end for
```

This approach achieves 0.55 Gflops per (8 cores) DSP. The TI compiler reports the number of cycles required to execute each iteration of each loop, allowing us to estimate the total number of required cycles assuming no memory stalls. Dividing this number by the actual number of execution cycles (recorded by a performance counter) we were able to determine that only 39.6% of cycles were spent on computing, meaning that 60.4% of the execution time was spent waiting for memory access.

B. Scratchpad Memory and Manual Unroll

A unique features of the C6678 is its on-chip memory architecture. Each core has a 32KB L1 cache and 512KB L2 that can be programmatically configured to behave as a traditional cache, as a program-controlled scratchpad memory, or as a combination of both. 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 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. When a kernel reads a data structure using a regular access pattern, this allows a block of onchip data can be processed on the DSP while the next block is read from off-chip memory. Likewise, when a kernel writes a data structure using a regular access pattern, this allows a block of onchip data to be rendered by the DSP while the previous block is being written to off-chip memory. This allows the DSP to overlap computation and communication in a way that is tailored to the software as opposed to blocking the instruction stream when a cache miss occurs. On the other hand, data structures that are read or written in an irregular pattern can still be cached and take advantage of locality.

In our first optimization, 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** are cached. This resulted in an increase to 0.66 Gflops per DSP and a reduction to 25.7% of the execution time waiting for memory.

Although the compiler supports automatic loop unrolling, we manually unrolled the loop by a factor of 2, 4, 8, and 16. We found that 8 gives the best performance and resulted in a

further improvement to 0.78 Gflops per DSP but an increase in memory waiting time to 28.8%.

C. Predicated Instructions and Assembly Optimization

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 IF statement on line 4 of Algorithm 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 the assembly language, which resulted in better utilization of the processor's VLIW utilization, increasing the performance to 1.63 Gflops per DSP while increasing the memory waiting time to 50.1%.

D. Loop Fission

Our next optimization was to fission the main loop into a product loop and an accumulate loop. Algorithm 2 shows the resulting implementation. In this case, the product loop is implemented in C while the accumulation loop is implemented in assembly language with a manually unrolled loop and predicated instructions. The algorithm is for each block of M entries fitting in the scratchpad SRAM.

Algorithm 2 Loop Fission

```
for i = 0 → M do //product loop
    prod[i] ← α × val[i] × x[col[i]]
end for
Acc ← 0
for i = 0 → M step by K do //accumulation loop
    Acc ← Acc + prod[i]
    if ptr[row] == i then
        row ← row + 1
        y[row] ← y[row] × β + Acc
        Acc ← 0
    end if
    Acc ← Acc + prod[i + 1]
    if ptr[row] == i + 1 then
        row ← row + 1
        y[row] ← y[row] × β + Acc
        Acc ← 0
    end if
...
    Acc ← Acc + prod[i + K]
    if ptr[row] == i + K then
        row ← row + 1
        y[row] ← y[row] × β + Acc
        Acc ← 0
    end if
end for
```

The product loop has no dependencies and can be software pipelined by the compiler, resulting in high computational performance, but 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.

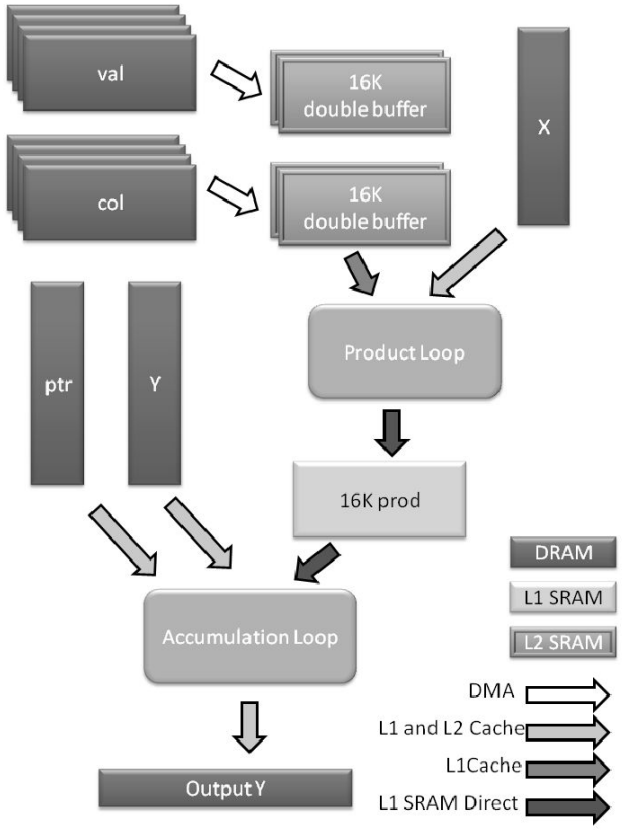

Fig. 1. Algorithm Overview

By removing the multiplies from the accumulation loop, performance are also gained from reduced register usage and data locality.

The loop fission resulted in a performance improvement to 2.08 Gflops per DSP and a drop in memory waiting time to 36.6%.

The Algorithm 2 and the memory arrangement are shown in Figure 1.

E. SIMD Instructions

The C6678 includes single instruction, multiple data (SIMD) instructions, allowing two single precision operations or load/stores to be performed by a single instruction. However, only two of the eight VLIW slots can be used for load/store instructions, so by using the SIMD instructions we can improve the memory bandwidth by a factor of two but the kernel is still memory bounded. Even so, adding SIMD instructions resulted in a 5% performance improvement to 2.2 Gflops.

F. Adaptive Row Pointer

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

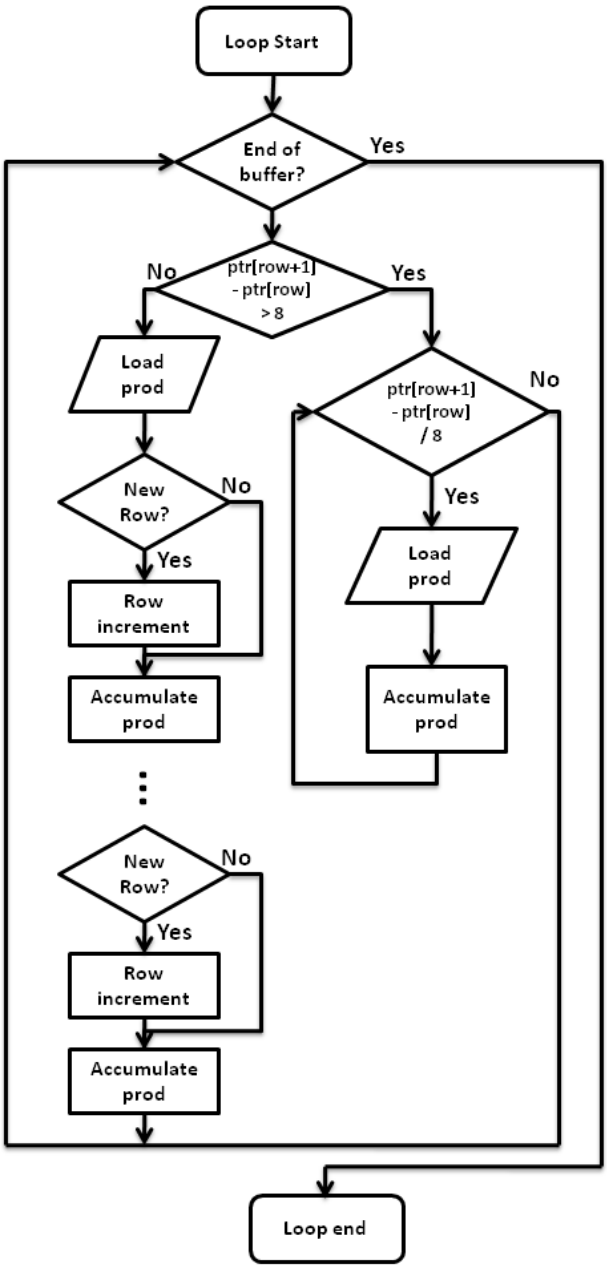

Fig. 2. Adaptive Accumulation Loop

to reduce this overhead, we added an optimization where, the start of the loop body, the code checks the number of values remaining on the current row, i.e. $ptr\,[row+1] - ptr\,[row]$.

As shown in Figure 2, 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 of the quick iteration. This optimization implies the more continuous values in a row, the better performance the kernel would achieve. Though the input matrices varies, generally, matrices with more non-zero values per row has more chances to benefit from it (Figure 4 and Figure 5).

IV. Performance Results

A. Experimental Setup

Using the same benchmark we performed an experiment that measured the power efficiency of the three test processor technologies. To collect the results we used a Yokogawa WT500 power analyzer to monitor the current and voltage at the AC wall socket connector. For the CPU we measured the difference in system power consumption when idle versus running the SpMV benchmark. For the GPU, we powered the GPU coprocessor card with a PCIe power cable from a secondary ATX power supply that is independent from the host's power supply and measured its power consumption when running the kernel. For the DSP we measured the power consumption of a standalone DSP evaluation board when running the kernel. Note that the CPU is at a disadvantage in these tests since its power usage includes that of the hard disks, network adapter, and video interface.

We have two reference GPU platform in this test, a high-end GTX680 and a main-stream GTX650Ti. Both of them are of NIVIDIA's latest Kepler architecture. To gather performance data for SpMV we used NVIDIA's CUSPARSE library. Our reference CPU is a four-core Core i5 650 CPU. To gather performance data for SpMV we used Intel's MKL library.

B. Power Efficiency

When using a tri-diagonal matrix (which contains three elements per row centered on the diagonal), a single eight-core DSP achieves 1.67 Gflops, while the Intel i5 650, NVIDIA GTX650Ti, and NVIDIA GTX680 achieves 1.89, 4.96, and 12.50 Gflops respectively. In Gflops per Watt, the DSP achieves 0.12, while these three platforms achieve 0.02, 0.17, and 0.21 respectively.

In order to test kernel performance with denser matrices, we scaled the tri-diagonal matrix by increasing the number of values per row, keeping the total number of elements unchanged which we refer to this as N-diagonal, where N is the number of values per row.

Figure 3 and Figure 4 show the performance and power efficiency of the four processors as the matrix density is scaled. The C6678 achieves equivalent or better power efficiency for denser matrices, of which $N > 71$ in Figure 4.

As well as the N-diagonal matrices, Figure 5 shows the power efficiency results for matrices from and the University of Florida Matrix Collection [7] and Matrix Market [8]. Notice that the normalized results relative to C6678 are shown in Figure 6.

In our experiment, some performance fluctuation is found in both the sample matrices and N-diagonal matrices for higher nnz/row. Generally, we expect matrices with more nonzero values per row to achieve better performance, since the program has more chance to go through the "easier" branch instead of the alternative with more predicated instructions (Figure 2).

V. Memory and Functional Unit Efficiency

Recall that SpMV performs the operation $Y = A\alpha X + \beta Y$. For single precision values and 32-bit indices, each non zero

TABLE I. MATRICES USED FOR TESTING AND CORRESPONDING POWER EFFICIENCY RESULTS

Matrix	Rows	Columns	Nonzeros	Nonzeros /Row	Intel i5 650	Nvidia GTX680	Texas Instruments C6678
TSOPF_FS_b300_c3	84414	84414	13135930	155.6	0.041	0.244	0.225
pdb1HYS	36417	36417	4344765	119.3	0.054	0.333	0.260
m_t1	97578	97578	9753570	99.9	0.048	0.290	0.256
audikw_1	943695	943695	77651847	82.3	0.033	0.181	0.278
consph	83334	83334	6010480	72.1	0.049	0.381	0.212
cant	62451	62451	4007383	64.2	0.052	0.372	0.160
pwtk	217918	217918	11524432	52.9	0.036	0.362	0.144
shipsec1	140874	140874	3568176	25	0.041	0.323	0.189
ldoor	952203	952203	23737339	24.9	0.031	0.302	0.155
lhr71c	70304	70304	1528092	21.7	0.038	0.286	0.149
thermal1	82654	82654	574458	6.9	0.015	0.215	0.143
mac_econ_fwd500	206500	206500	1273389	6.1	0.019	0.278	0.137
ASIC_100ks	99190	99190	578890	5.8	0.024	0.215	0.129
scircuit	170998	170998	958936	5.6	0.021	0.269	0.114
shyy161	76480	76480	329762	4.3	0.025	0.221	0.082
mc2depi	525825	525825	2100225	4.0	0.026	0.277	0.096

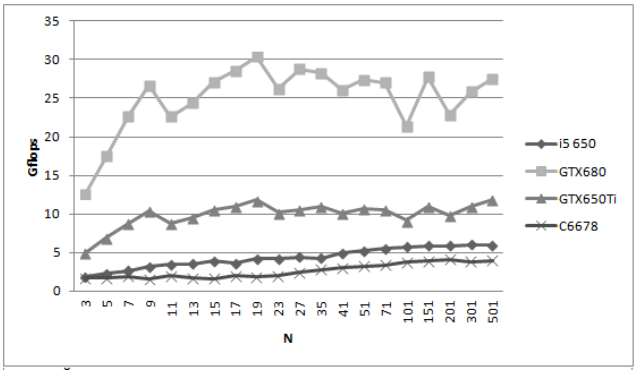

Fig. 3. SpMV Performance on N-diagonal Matrices

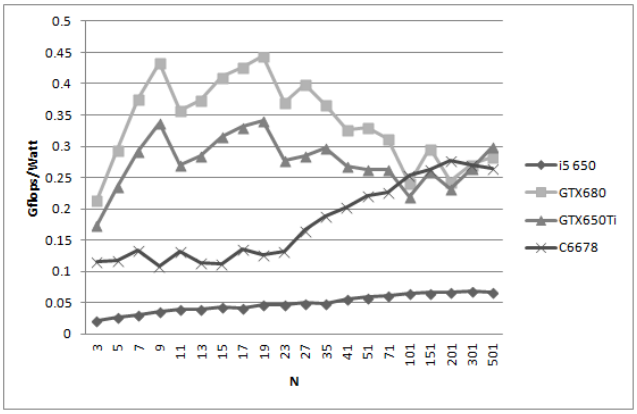

Fig. 4. SpMV Gflops/Watt on N-diagonal Matrices

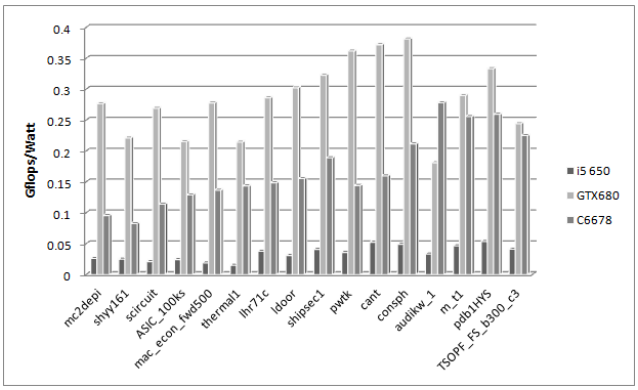

Fig. 5. SpMV Performance on Selected Matrices (matrices are listed on the x-axis in ascending order in terms of nonzeros/row)

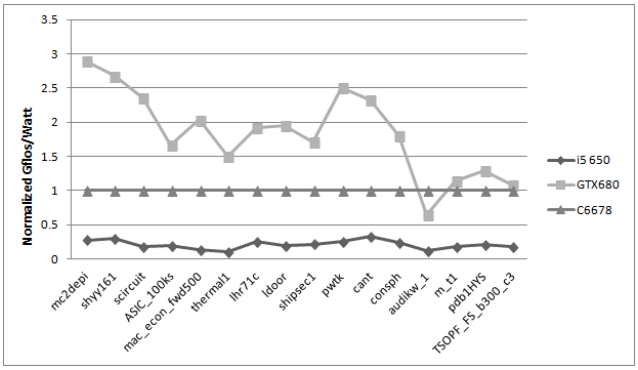

Fig. 6. Normalized SpMV Performance (matrices are listed on the x-axis in ascending order in terms of nonzeros/row)

value in A, SpMV performs two multiplies (with α and X) and one add (to compute the dot product), which requires a load of a four-byte value, a four-byte column index, and a four-byte vector value from X (is a scalar). If we assume the cache is perfect and all the values in X would be referred at least one time during computation, the 3ops in SpMV kernel would require $8+4/row$ bytes due to compulsory misses. For each matrix row, the kernel performs an additional multiply (with β) and add (to Y), which requires a four-byte load from Y, a four-byte store back to Y, and a four-byte load from the ptr array. Row operations occur at frequency of n-times less often than value operations, where n = average number of non-zero elements per row. As such, SpMV has an arithmetic intensity(AI) of floating point operations per byte is shown as Equation 2:.

$$AI = (1 \times \frac{3ops}{8 + \frac{4}{rows}bytes} + \frac{1}{n} \times \frac{2ops}{12bytes})/(1 + \frac{1}{n}) \quad (1)$$

$$= \frac{9 \times rows \times n + 8 \times rows + 2}{12(2 \times rows \times n + n + 2 \times rows + 1)} ops/byte \quad (2)$$

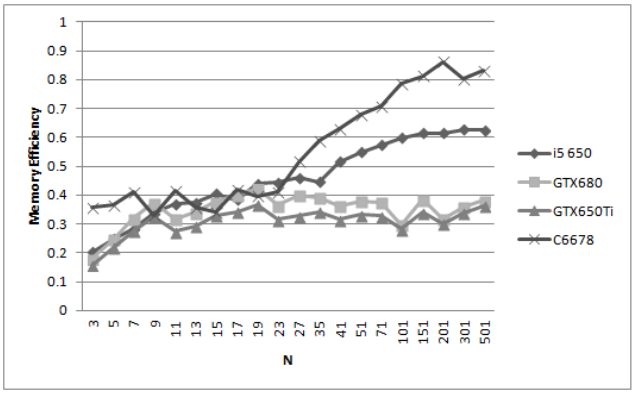

Fig. 7. Memory Efficiency on N-diagonal Matrices

TABLE II. PEAK AND ACTUAL PERFORMANCE

	Intel i5 650	Nvidia GTX680	Nvidia GTX650Ti	Texas Instruments C6678
Max memory throughput	25.6 Gbytes/s	**192.3** Gbytes/s	86.4 Gbytes/s	12.8 Gbytes/s
Peak computational throughput	9.59 Gflops	**72.1** Gflops	32.4 Gflops	4.80 Gflops
Actual performance	5.89 Gflops	**27.8** Gflops	11.0 Gflops	3.9 Gflops
Memory efficiency	0.61	0.39	0.34	**0.81**

For most modern architectures, this level of arithmetic intensity makes SpMV a memory-bound operation. This allows us to compute the peak computational throughout as the product of arithmetic intensity and peak memory bandwidth. The C6678 has a peak memory bandwidth of 12.8 Gbytes/s, giving a peak computational throughput of $12.8 \times AI$ Gflops, while the NVIDIA GTX 680 has a peak memory bandwidth of 192.3 Gbytes/s, giving a peak computational throughput of $192.3 \times AI$ Gflops.

Table II shows the peak vs. actual computational throughput for the diagonal matrix with 151 entries per row and 208,326 rows. Memory efficiency is calculated as the ratio of these values. As shown, the C6678 achieves the highest memory efficiency of all three architectures.

More memory efficiency results are shown in Figure 7. After $N = 27$ the DSP achieves higher memory efficiency than the CPU and GPUs.

VI. COMPUTATIONAL EFFICIENCY

Peak computational capacity is often computed by multiplying the number of functional units by the clock rate. Using this metric, the C6678 has a peak throughput of 128 Gflops while the NVIDIA GTX 680 has a peak throughput of 3090 Gflops, thus for the diagonal matrix with 151 entries per row the C6678 achieves 3% of its peak performance while the NVIDIA GTX 680 achieves 0.9% of its peak performance.

Since the floating-point operations in SpMV with CSR depend on memory and integer operations, we can also measure the peak computational rate by measuring the throughput of our SpMV kernel by restricting the input to only on chip memories. In other words, in order to explore the degree in which the memory bound reduces overall performance for the DSP, we did an experiment in which we only measured the execution time required to process a block of data that was already loaded into an on-chip buffer. For the diagonal matrix with 151 entries per row, this on chip performance is 4.4 Gflops. The actual performance, 3.9 Gflops, is 88.6% of the on-chip performance, indicating a good balance between memory bandwidth and on-chip capacity.

VII. RELATED WORK

Optimizing sparse matrix-vector multiply on emerging architectures has been the subject of much recent work [9][10][11][12]. To our best knowledge, this is the first implementation of SpMV on the current generation of the TI DSP architecture. There has been some recent work in dense linear algebra on this architecture, demonstrating 80 single precision Gflops for SGEMM[13]. Although the authors of this paper didn't directly measure the DSP's power consumption, manufacturer data suggests the device worst case consumption is 10 Watts, indicating the possibility (though not yet directly measured) of 8 Gflops/Watt for SGEMM.

There has also been recent interest in predicting and minimizing power consumption for GPUs [14][15][16][17][18][19]. Improving the performance or power efficiency of SpMV on GPUs has also been an area of intense study [20][21][22][23][12][24][25][26].

VIII. CONCLUSION

The Texas Instruments C6678 is a substantially smaller device than a typical CPU or graphical processor unit but is still able to deliver impressive floating-point performance. As such, the architecture has potential as a building block for massively parallel arrays of these devices, assuming they achieve high power efficiency. In this paper we demonstrated that the C6678 is capable of achieving higher power efficiency than a CPU and equivalent power efficiency than a GPU despite having the disadvantage of being manufactured in 45 nm process versus 32 nm and 28 nm process for the CPU and GPU, respectively. We assume that the TI CorePac architecture, when implemented in more advanced fabrication processes, will significantly exceed the power efficiency of the state-of-the-art GPUs for some scientific kernels.

ACKNOWLEDGMENT

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 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 & Engineering*, pp. 8–10, 2010.

[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. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, 2010.

[5] R. Coifman, V. Rokhlin, and S. Wandzura, "The fast multipole method for the wave equation: A pedestrian prescription," *Antennas and Propagation Magazine, IEEE*, vol. 35, no. 3, pp. 7–12, 1993.

[6] K. K. Nagar and J. D. Bakos, "A sparse matrix personality for the convey hc-1," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*. IEEE, 2011, pp. 1–8.

[7] T. A. Davis, "The university of florida sparse matrix collection," in *NA digest*. Citeseer, 1994.

[8] R. F. Boisvert, R. Pozo, K. Remington, R. Barrett, and J. J. Dongarra, "Matrix market: a web resource for test matrix collections," *Quality of Numerical Software, Assessment and Enhancement*, pp. 125–137, 1997.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[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] K. K. Kasichayanula, "Power aware computing on gpus," Master's thesis, University of Tennessee, 2012.

[15] R. Suda *et al.*, "Accurate measurements and precise modeling of power dissipation of cuda kernels toward power optimized high performance cpu-gpu computing," in *Parallel and Distributed Computing, Applications and Technologies, 2009 International Conference on*. IEEE, 2009, pp. 432–438.

[16] J. W. Sheaffer, D. Luebke, and K. Skadron, "A flexible simulation framework for graphics architectures," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. ACM, 2004, pp. 85–94.

[17] X. Ma, M. Dong, L. Zhong, and Z. Deng, "Statistical power consumption analysis and modeling for gpu-based computing," in *Proceeding of ACM SOSP Workshop on Power Aware Computing and Systems (HotPower)*, 2009.

[18] S. Collange, D. Defour, and A. Tisserand, "Power consumption of gpus from a software perspective," *Computational Science–ICCS 2009*, pp. 914–923, 2009.

[19] J. M. Cebri'n, G. D. Guerrero, and J. M. Garcia, "Energy efficiency analysis of gpus," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012, pp. 1014–1022.

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

[21] 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.

[22] 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.

[23] 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.

[24] F. Vazquez, G. Ortega, J. Fernandez, and E. 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.

[25] 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*, pp. 1–9, 2011.

[26] H. Anzt, V. Heuveline, J. I. Aliaga, M. Castillo, J. C. Fernandez, R. Mayo, and E. S. Quintana-Orti, "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.