Evaluating the Potential of Graphics Processors for High Performance Embedded Computing

Shuai Mu[1], Chenxi Wang[1], Ming Liu[2], Dongdong Li[2], Maohua Zhu[1], Xiaoliang Chen[3], Xiang Xie[1], Yangdong Deng[1]

[1]Tsinghua University, [2]BeiHang University, [3]Chinese Academy of Sciences

{mus04ster, zhumaohua, yangdong.deng, lmingcsce}@gmail.com, wcx09@mails.tsinghua.edu.cn, xiexiang@tsinghua.org.cn,

Abstract—**Today's high performance embedded computing applications are posing significant challenges for processing throughout. Traditionally, such applications have been realized on application specific integrated circuits (ASICs) and/or digital signal processors (DSP). However, ASICs' advantage in performance and power often could not justify the fast increasing fabrication cost, while current DSP offers a limited processing throughput that is usually lower than 100GFLOPS. On the other hand, current multi-core processors, especially graphics processing units (GPUs), deliver very high computing throughput, and at the same time maintain high flexibility and programmability. It is thus appealing to study the potential of GPUs for high performance embedded computing. In this work, we perform a comprehensive performance evaluation on GPUs with the high performance embedded computing (HPEC) benchmark suite, which consist a broad range of signal processing benchmarks with an emphasis on radar processing applications. We develop efficient GPU implementations that could outperform previous results for all the benchmarks. In addition, a systematic instruction level analysis for the GPU implementations is conducted with a GPU micro-architecture simulator. The results provide key insights on optimizing GPU hardware and software. Meanwhile, we also compared the performance and power efficiency between GPU and DSP with the HPEC benchmarks. The comparison reveals that the major hurdle for GPU's applications in embedded computing is its relatively low power efficiency.**

Keywords: GPU, Multi-core, HPEC benchmark, DSP, parallel computing, Fermi, GFLOPS

I. INTRODUCTION

Today people's living quality is largely determined by the underlying information technology (IT) infrastructure. With the increasing number of IT service users and diversifying customer requirements, the IT infrastructure has to deliver significant computing power and thus poses the demand for high performance embedded computing. Typical applications in this category include Internet routing, cellular wireless base station, and radar/sonar. For instance, now the core Internet routers have to achieve a throughput of 40Gbps to 90Tbps [1]. And a 4G wireless base station needs to provide a 1Gbit/s data rate for a (relatively fixed) customer, with up to 200 active subscribers in the service area [2].

Traditionally, the abovementioned high performance embedded computing applications depend on application specific integrated circuits (ASICs) to realize the functionality. ASICs offer the highest performance and best power-efficiency, but they are inflexible for upgrading and customization. With the fast increasing fabrication cost as well as the ever-growing requirements for service differentiation, DSPs have gradually become the workhorse for performance-demanding embedded systems. However, DSPs are initially designed for low-power and low-cost applications. Thus, current single-chip DSPs usually could only deliver relatively limited computing power. For example, TI's high performance embedded processor, TMS320C6472 [3], has 6 cores and achieve a peak performance of 67.2GFLOPS [4]. To provide a sufficient level of performance, a high performance embedded system usually has to be assembled with multiple boards with each installing one or more DSPs. Such multi-board solutions tend to be bulky and costly due to the extra levels of packaging. Moreover, the distribution of program data over multiple DSPs would also incur performance overhead.

On the other hand, modern multi-core general-purpose processors have become very powerful computing machines. Especially, current GPUs could attain a peak float-point throughput of up to 1536GFLPOS [5], which is higher than that of the fastest DSP by a factor of over 20. In addition, GPU follows a single program multiple data (SPMD) programming model [6], which well matches the underlying computing patterns of most high performance embedded applications.

Accordingly, it is appealing to investigate the potential of GPUs for high performance embedded computing. In this paper, we quantitatively analyze the performance implications of GPU and DSP on the high performance embedded computing (HPEC) benchmark suite, which was developed by MIT High-Productivity Computing Systems (HPCS) program [7]. The benchmark suite consists of a set of benchmarks covering a broad range of advanced signal processing applications with an emphasis on the radar processing. The major contributions of this work include the following.

- To the best of our knowledge, this work is the first one to implement the complete HPEC benchmark suite on GPUs. Our GPU solutions achieve better performance than all previously published results using graphics processors. Especially, our GPU based QR factorization and singular value decomposition procedures could significantly outperform their CPU equivalents. The techniques developed in this work would be essential for many embedded and scientific applications.
- A detailed analysis of the characteristics of HPEC benchmarks is performed. Such essential characteristics as parallel granularity, instruction classification, branch divergence among synchronized threads, as well as memory access patterns were extracted by using a research GPU microarchitecture simulator (i.e., GPGPU-

978-3-9810801-7-9/DATE11/©2011 EDAA

Sim) [8]. The characterization would provide key insight to optimizing GPU microarchitecture for high performance embedded computing.

- To the best of authors' knowledge, this is the first work to develop a quantitative performance comparison among CPU, GPU and DSP on the HPEC benchmarks.

The rest of this paper is organized as follows. In section 2, we briefly review the related work. An overview of HPEC benchmarks and GPU architecture will be given in section 3. In section 4, we implement GPU solutions and compare the results with CPU and DSP. Section 5 gives a detailed analysis to GPU implementations. Section 6 concludes the paper.

II. Related Work

Many signal and image processing algorithms in embedded applications have been implemented on GPUs. An incomplete list of these works include fast Fourier transform [9, 10], convolution [11], pattern recognition and vision algorithm (e.g., tracking and stereo reconstruction [12]). These works focus on a specific problem. However, high performance embedded systems generally involve a set of application, and the acceleration of one application does not necessarily suggest the performance advantage of the whole system. Both Pettersson [13] and Kerr [14] implemented a subset of HPEC benchmarks on GPUs and reported superior results to CPU solutions. However, implementation details have not been reported. Our work provides a systematic investigation by implementing the whole HPEC benchmark suite on the latest NVidia GPU. In addition, we conducted a comprehensive characterization for GPU execution of the HPEC benchmarks. A performance and power comparison with DSP is also performed.

III. Overview

In this section, we review the HPEC benchmarks as well as the NVidia GPU architecture.

A. HPEC Benchmarks

The HPEC benchmark suite defines nine kernel programs identified through a survey of a broad range of signal processing application areas such as radar and sonar processing, infrared sensing, signal intelligence, communication, and data fusion. The functionality and characteristics of these benchmarks are listed in Table I and Table II, respectively.

TABLE I. Benchmarks Descriptions

Benchmark	Description
TDFIR	Time-domain finite impulse response filtering
FDFIR	Frequency-domain finite impulse response filtering
CT	Corner turn via matrix operations
PM	Pattern matching
CFAR	Constant false-alarm rate detection
GA	Genetic algorithm
QR	QR factorization
SVD	Singular value decomposition
DB	Database operations

Each benchmark has several data sets with varying scales to compute in Table II. The task-level parallelism indicates that the number of independent tasks that could be executed concurrently. The workload is derived through a sequential execution on a CPU. Both above parameters are from the performance statistics released with HPEC. The underlying data structures include vector, matrix, and tree. Therefore, the corresponding data size refers to vector length, matrix dimension or number of tree nodes. The data correlation reflects the dependence between different elements in the same task. The memory access column shows the relative intensity of communication compared with computation.

TABLE II. HPEC Benchmarks Properties

Benchmark	Data Set	Workload (MFLOP)[1]	Task-Level Parallelism	Data Structure	Data size	Data Correlation	Memory access
TD FIR	Set 1 Set 2	268.4 1.97	64 20	Vector	4096 1024	Low	Low
FD FIR	Set 1 Set 2	34 2.21	64 20	Vector	4096 1024	Low	Low
CT	Set 1 Set 2	2 30	1 1	Matrix	50x5000 750x5000	No	Very High
PM	Set 1 Set 2	1.21 13.59	72 256	Vector	64 128	Low	Low
CFAR	Set 1 Set 2 Set 3 Set 4	0.17 150.5 41.1 17.7	384 6144 3072 480	Vector	64 3500 1909 9900	Medium	Low
GA	Set 1 Set 2 Set 3 Set 4	0.011 0.51 0.015 0.11	50 200 100 400	Vector	8 96 5 10	Medium	High
QR	Set 1 Set 2 Set 3	397 30.5 45	1 1 1	Matrix	500x100 180x60 150x150	High	Medium
SVD	Set 1 Set 2	0.24 0.88	1 1	Matrix	500x100 180x60	High	Medium
DB	Set 1 Set 2	440 700	1 1	Tree	440 700	High	Very High

B. Overview of GPU Computing

In this work, the GPU programs were implemented on a Fermi GPU, which is NVidia's latest generation graphics processor. Delivering a 1536GFLOPS single-precision float point performance, a Fermi GPU installs 14 stream multiprocessors (SM), while a SM has 32 streaming processors (SP).

The GPU program follows a SPMD model with a thread as the basic unit for parallel computations. Tens of thousands of threads can be launched. All these threads execute the same program, but on different data. The threads are first organized into a block in which threads could synchronize and share data in a fast on-chip memory. Threads in different blocks could only share data in the global memory of GPU.

During execution, every 32 threads in a block (called a warp) would follow exactly the same instruction schedule. If these threads take different conditional paths, a performance overhead would be incurred. Note that a function called by CPU but executed on GPU is designated as a kernel.

IV. Implementation on GPUs

In this section, we introduce the GPU implementations of the HPEC benchmarks. We also compare the performance between GPU and DSP implementations.

[1]The workload of CT and DB are measured in MB and Transactions, respectively.

A. Implementation

We take advantage of both task level and data level parallelism when developing GPU based implementations. For those benchmarks with available task level parallelism, different tasks could be assigned to different blocks. The communication and synchronization between different blocks will be realized by launching new kernels. We also performed extensive optimizations using such techniques as maximizing the usage of shared memory, making global memory accessing coalesced and reducing program divergence.

Time-Domain Finite Impulse Response (TDFIR) kernel assigns one element of the input vector to each thread. During each accumulation, all the elements of input vector are multiplied by the same filter coefficient but summed up to different elements of output vector.

Frequency-Domain Finite Impulse Response (TDFIR) kernel can readily realized by calling the CUFFT Library [15].

Corner Turn (CT) actually performs a transposition operation on a matrix. We followed an implementation presented in CUDA SDK [16]. The key idea is to use shared memory to eliminate un-coalesced global memory access and shared memory bank conflict.

Pattern Matching (PM) kernel is extracted from the feature-aided tracking portion of an integrated radar-tracker application [17]. It entails overlaying two length-N vectors A and T and then computes a metric that quantifies the degree to which these two vectors match each other. In general, the vector T is chosen from a set of reference vectors referred to as the template library. We map one reference vector to one block and compute the similarity with other vectors simultaneously.

Constant False-Alarm Rate (CFAR) detection benchmark is an example of data-dependent processing. The objective is to find targets in an environment of varying background noise. This problem could be simplified to find an abnormally large element in a data vector. To judge whether an element is large enough, we should first calculate the average value of its neighboring entries and then perform a comparison. The parallelization is straightforward by checking multiple elements in parallel.

Genetic Algorithm (GA) is a statistical optimization mechanism to search the global optimum through a series of local searches. During the optimization process, each candidate solution is designated as a "chromosome" consisting of the basic elements of "genes". There have been a few publications (e.g., [18-20]) on implementing GA on GPUs. Typically, different candidate solutions are assigned to different thread blocks. At each generation, the selection, crossover, mutation and evaluation operations are performed serially. Several key points should be resolved properly to attain an efficient GA implementation. The first concern is a parallel random generator. The GA performance is largely determined by the quality of pseudo random numbers, which is essential to have a parallel generator closely matching the uniform distribution. We adopt the common linear congruence algorithm [21] and make every thread maintain its dedicated generator with seeds for uniform distribution. Another concern is the communication between blocks. During selection and evaluation, the candidate solutions have to exchange elements very frequently. The traffic certainly worsens the pressure on memory accessing. The on-chip cache and the user-programmed share memory are playing an important role to hide the memory latency.

QR factorization (QR) is a linear algebra operation that factors a matrix into an orthogonal component, Q, and an upper triangular component, R. The QR benchmark in HPEC implements a fast Givens QR factorization algorithm [22], which is hard to extract parallelism effectively because the computation in each step depends on data derived in the previous iteration. Although several works [23, 24] tried to tame such irregular data operations, their implementations focused on matrix-matrix multiplication and the Householder algorithm rather than fast Givens. In this work, we found that the pipeline parallelism is rich in the fast Givens algorithm. In fact, the Givens rotation and matrices updating are executed from the bottom row to the diagonal in a column-by-column manner, while the computation of higher rows depends on the results of lower adjacent rows. Given a pipelined scheduling, the accessing of higher rows can be performed with the computing of lower rows in parallel. The synchronization between different blocks is realized through kernel initialization. The pipelined scheduling implementation offers a notable improvement of nearly 4X speed-up compared to previous results [14]. The throughput is even higher than that of GPU based Householder algorithm [23].

Singular Value Decomposition (SVD) is an advanced linear algebra operation with wide signal processing applications. A GPU based SVD implementation was proposed in [25], although it uses a different algorithm from the one in HPEC benchmark. An analysis on the HPEC SVD benchmark indicates that the QR factorization and matrix multiplication operation consume over 50% computing time of SVD kernel. Therefore, we use our optimized QR kernel and matrix multiplication kernel of CUDA SDK [16] to solve the SVD problem and observe considerable performance improvements.

Database (DB) operations present unique challenges due to its irregular data structures, especially the binary tree structure. In fact, the basic tree operations such as insert, delete and select incur a great amount of memory accessing, but little computations. It is extremely difficult to find an efficient GPU implementation for such a computing pattern. In [26], the authors replaced the binary tree with a linear array. However, this approach only expedited the select operation, but had little impacts on the efficiency of insert and delete operations. To minimize the impact of memory latency and exploit thread-level parallelism, Kim et al. [27] presented a fast software framework for index tree manipulations by considering microarchitecture features like page size, cache line size and SIMD width. In this work, the original binary tree implementation turns to be quite

inefficient for GPU. Therefore, we use hash function to store the database as such a scheme has lower memory bandwidth demand at an increased computing density. The updating of hash tables is also less complex than the original tree structure on GPUs. To reduce the computing complexity, we choose a hash function purely composed of shifts and bitwise operations, i.e., (f(x) = x ^ (x>>10) ^ (x>>20)). As a result, the operations of insert and delete could be executed in parallel if they access different positions of hash tables. It should be noted that the GPU throughput for DB is still limited because DB operations are inherently not friendly to data-parallel machines. On the other hand, our technique is orthogonal to that proposed in [27] and it is appealing to combine the two approaches to pursue a newer level of performance.

B. *Results*

We conducted extensive experiments to evaluate the performance of GPU implementations. All experiments are performed on a Linux PC with a 2.66-GHz, Core 2 Duo processor and an NVidia Tesla C2050 graphics card. The graphics card has 3GB memory. All programs are compiled with CUDA release 3.1 [16]. The performance results are shown in Table III. We compare the performance between CPU and GPU platforms in terms of floating-point throughput, i.e., GFLOPS, except that CT is measured in GB/s and DB is rated in million transactions per second. When calculating the final throughput, the transferring time between CPU and GPU are excluded for two reasons: 1) we want to focus on the pure computing throughput of different platforms; and 2) in this work, we evaluate the GPU's potential for embedded systems where the memory transfer time could be overlapped with the computation time.

TABLE III. PERFORMANCE RESULTS

Kernels	Data set	CPU Throughput (GFLOPS) [2]	GPU Throughput (GFLOPS) [2]	Speedup
TDFIR	Set 1	3.382	97.506	28.8
	Set 2	3.326	23.130	6.9
FDFIR	Set 1	0.541	61.681	114.1
	Set 2	0.542	11.955	22.1
CT	Set 1	1.194	17.177	14.3
	Set 2	0.501	35.545	70.9
PM	Set 1	0.871	7.761	8.9
	Set 2	0.281	21.241	75.6
CFAR	Set 1	1.154	2.234	1.9
	Set 2	1.314	17.319	13.1
	Set 3	1.313	13.962	10.6
	Set 4	1.261	8.301	6.6
GA	Set 1	0.562	1.177	2.1
	Set 2	0.683	8.571	12.5
	Set 3	0.441	0.589	1.4
	Set 4	0.373	2.249	6.0
QR	Set 1	1.704	54.309	31.8
	Set 2	0.901	5.679	6.3
	Set 3	0.904	6.686	7.4
SVD	Set 1	0.747	4.175	5.6
	Set 2	0.791	2.684	3.4
DB	Set 1	112.3	126.8	1.13
	Set 2	5.794	8.459	1.46

The experiment results in Table III can only be fully understood by considering the benchmark characteristics shown in Table II. Overall, the problem scales for all the benchmarks are not very large. Our GPU implementations always perform better on large data set. Compared with the CPU results, GPU could receive an order of magnitude speedup on average. Specifically, GPU performs quite well when an application contains rich task-level parallelism (e.g., TDFIR, FDFIR, and CFAR) or could be partitioned into independent blocks of threads that do not require fine-grain communication and synchronization (e.g., CT and PM). Although GA has plenty of task level parallelism, the data level parallelism is so low (each block has no more than 10 threads for 3 data sets) that there are not enough active threads in each block to hide the memory accessing time. Therefore, their performance drops significantly. Applications containing limited task parallelism (e.g., QR and SVD) could extract more data level parallelism using the pipelined scheduling. DB application involves many random global memory accesses, which dramatically reduces the bandwidth efficiency while the cache architecture could partially make up for the random accessing to increase the bandwidth efficiency. Overall, compared to previous work [14], our GPU implementations are superior for almost all the benchmarks. Note that related previous work, i.e., [14], did not discuss SVD and DB.

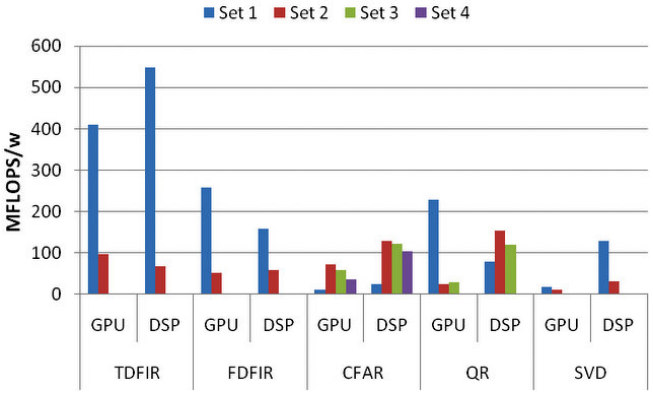

Figure 1. Power Efficiency

C. *Comparsion with DSP*

We also compared the performance of HPEC benchmarks between GPU and DSP platform. In our experiment, we use ADSP-TS101S TigerSHARC 6U cPCI [28] board that contains eight DSPs. Each DSP has a peak floating point performance of 1.5GFLOP. This platform could deal with multi-channel input data, but we only use one channel. Due to the limitations of DSP architecture, we only implemented 5 benchmarks, TDFIR, FDFIR, CFAR, QR, and SVD on DSP. For all benchmarks, GPU could outperform DSP by two orders of magnitude.

On the other hand, it is interesting to compare the power efficiency between GPU and DSP because power is becoming an increasingly important issue for embedded application. Figure 1 gives the comparison of throughput per watt for both GPU and DSP platforms. The measured power consumptions of ADSP-TS101 and Tesla C20250 are 10w and 238w, respectively.

Our results reveal that DSPs do offer a considerably better power efficiency. It should be noted that the ADSP-TS101 is not the fastest available DSP. When compared to

[2]The throughputs of CT and DB are measured in GB/s and Million Transactions/s.

high-end DSPs (e.g., TI TMS320C6472-700), the gap in power efficiency can be even greater.

V. DETAILED ANALYSIS

In this section, we present a detailed instruction analysis on the GPU implementations of the HPEC benchmark. The analysis helps understand why those benchmarks with abundant task-level or data-level parallelism do not achieve better performance. Specifically, we want to answer the following question: What is the performance bottleneck for a specific benchmark, thread parallelism, memory bandwidth or instructions branch? The analysis, on one hand, will point out the directions of fine-tuning the GPU implementations. On the other hand, it helps us identify the micro-architecture features to make GPU more suitable for embedded applications.

NVidia releases a profiling tool, CUDA profiler [16], for performance analysis. However, the statistics collected are not complete and detailed. Therefore, we use a cycle-accurate micro-architecture simulator GPGPU-Sim that has a similarity coefficient of 0.899 when compared with the real GPU hardware [29].

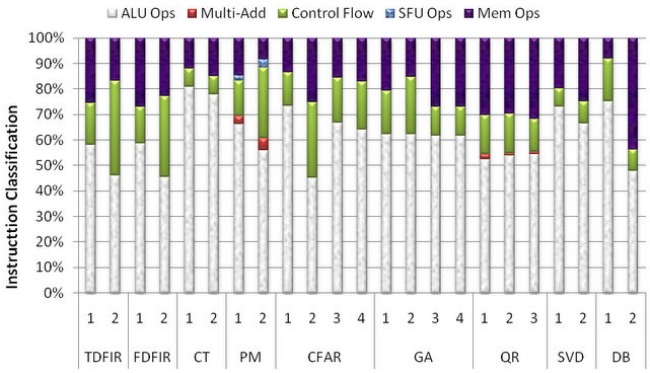

Figure 2. Instruction Classification

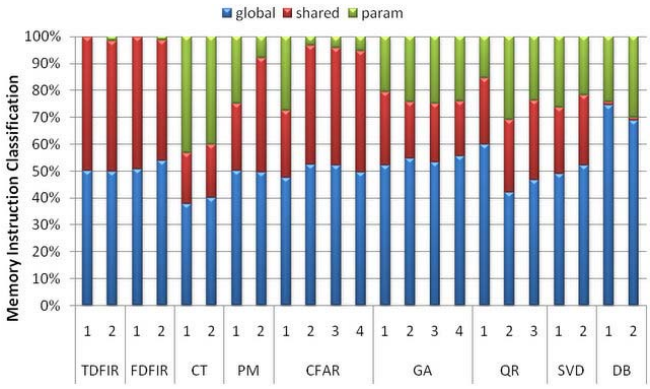

Figure 3. Memory Instruction Breakdown

A. Instructions Analysis

The decompositions of instructions according to their types are depicted in Figure 2 for each benchmark. The Multi-Add and ALU Ops sections of each bar show the proportion of total ALU operations. The SFU Ops instructions are executed by the special function units installed inside each SM. In Figure 3, memory operations are further broken down into three types.

In general, the ALU operations occupy more than half of total operations, although there do exist memory-intensive kernels (CT, DB). PM uses some SFU and MAC operations, which actually reduces the performance because the number of SFUs (4 in every SM) is far less than SPs. By correlating Figures 2, 3 and with Table III, the following trends can be identified.

- More ALU operations bring in better performance.
- The increasing usage of shared memory consistently leads to better performance.
- A larger percentage of control flow operations do not reduce the performance directly.
- Memory-bound kernels (CT, DB) do not mean there are more memory operations instructions. The performance is to a larger extension restricted by regularity of data access patterns.

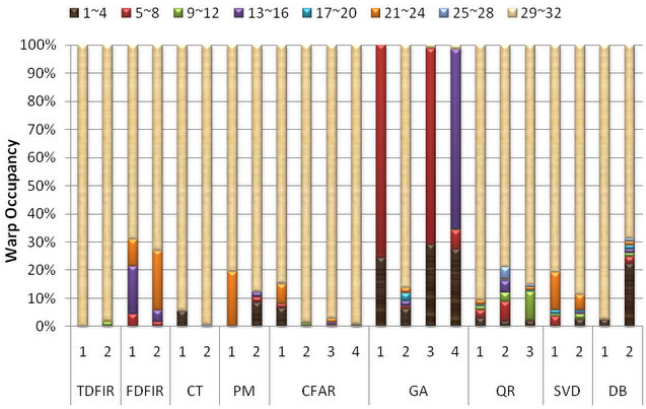

Figure 4. Warp Occupancy

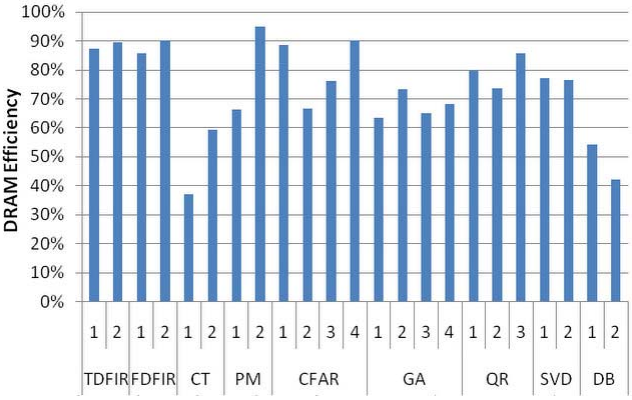

Figure 5. DRAM Efficiency

B. Warp Occupancy

Figure 4 shows warp occupancy values (number of active threads in an issued warp) over the entire runtime of the benchmarks. This metric can be used to evaluate how much potential GPU throughput is wasted due to unfilled warps. In Figure 2, the kernels of TDFIR, FDFIR have up to 30% control flow operations, while their warp occupancy is very high. The reason is that they have little warp-level branch divergence so that the computing is highly homogeneous. The branch divergence is a major stumbling obstacle for DB and GA, where more than 70% warps have less than 10 active threads.

C. Memory Bandwidth Utilization

Although the Fermi GPU has very high peak memory bandwidth of 230GB/s, an irregular memory-accessing pattern could still dramatically drags down the memory bandwidth utilization. It could happen that the whole SP pipelines have to be stalled for all warps. Figure 5 uses the concept of DRAM efficiency to evaluate the memory bandwidth usage quantitatively. DRAM efficiency is the percentage of time spent on sending data across the pins of DRAM over the time when there are any memory requests being serviced or pending in the memory controller's input buffer. Almost all benchmarks reach over 50% DRAM efficiency. We observe that high DRAM efficiency leads to high computing performance for most kernels. TDFIR, FDFIR and CFAR could achieve about an efficiency level of 90% as they depend on linear arrays and dense matrices. DB has a low efficiency due to its irregular memory accessing. One special phenomenon is that the efficiency of CT kernel is lower than other kernels. One possible reason is that our data set is too small.

VI. Conclusion and Future Work

As a first step toward systematically exploiting the potential of GPU for embedded computing, we developed efficient implementations of the HPEC benchmarks on NVidia's new generation GPU. Through careful orchestrating of various types of parallelism, our GPU implementations achieve superior results. The encouraging results justify the potential of GPUs for high performance embedded computing applications. A detailed instruction analysis was followed to characterize the GPU execution for HPEC benchmarks. The analysis provides key insight for optimizing data parallel algorithms on GPU architecture.

A comparison of power efficiency between GPU and DSP was also conducted. The results suggest that the big obstacle to apply GPU to embedded applications is its low power efficiency. We are currently working on developing micro-architectures that could combine the advantages of GPU and DSPs. Another important research direction is to reduce the GPU power consumption through optimizing the memory hardware and using adaptive techniques to match the required computing intensity automatically.

References

[1] H. J. Chao and B. Liu, "High Performance Switches and Routers," Wiley-Interscience, A John Wiley & Sons, INC., Publications, 2007.

[2] S. Glisic and J. P. Makela, "Advanced Wireless Networks : 4G Technologies", IEEE Symposium on Spread Spectrum Techniques and Applications, pp. 442-446, Aug. 2006.

[3] TI, High Performance DSP, TMS320C647x Multicore DSP http://focus.ti.com/docs/prod/folders/print/tms320c6472.html.

[4] Wikipedia, FLOPS, http://en.wikipedia.org/wiki/FLOPS.

[5] NVidia Fermi Compute Architecture Whitepaper, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[6] NVidia, CUDA Programming Guide for CUDA Toolkit 3.1.1, http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf.

[7] HPEC Challenge Benchmark Suite. http://www.ll.mit.edu/HPECchallenge/

[8] GPGPU-Sim, http://www.ece.ubc.ca/~aamodt/gpgpu-sim/.

[9] T. Jansen, B. R. Lipinski, N. Hanssen, and E. Keeve, "Fourier volume rendering on theGPU using a split-stream-FFT," in Proc. Vision,Modeling and Visualization Workshop, Stanford, pp. 395–403, Nov. 2004.

[10] N. K. Govindaraju, S. Larsen, J. Gray, and D.Manocha, "A memory model for scientific algorithms on graphics processors," in Proc. SuperComputing, pp. 6-6, 2006.

[11] Fialka, O. Cadik, "FFT and Convolution Performance in image Filtering on GPU", in Information Visualization, pp. 609-614, England, 2006.

[12] S. M. Seitz, B. Curless, J. Diebel, D. Scharstein, and R. Szeliski, "A comparison and evaluation of multi-view stereo reconstruction algorithms," in Proc. IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR), New York, vol. 1, pp. 519–526, Nov. 2006.

[13] J. Pettersson, "Radar Signal Processing with Graphics Processors (GPUs)", Master Thesis, Uppsala University, 2010. http://www.hpcsweden.se/files/RadarSignalProcessingwithGraphicsProcessors.pdf.

[14] A. Kerr, D. Campbell, M. Richards, "GPU Performance Assessment with the HPEC Challenge", HPEC Workshop in MIT Lincoln Laboratory, 2008.

[15] CUDA FFT Library, www.nvidia.com/object/cuda_develop.html

[16] CUDA SDK, http://developer.nvidia.com/object/cuda_3_1_downloads.html.

[17] W. Coate and M. Arakawa, "Preliminary design review: Feature-aided tracking for the PCA integrated radar-tracker application", Project Report PCA-IRT-5, MIT Lincoln Laboratory, MA, October 2004.

[18] J. Ming, X. J. Wang, R. S. He, Z. X. Chi, "An Efficient Fine-grained Parallel Genetic Algorithm Based on GPU-Accelerated", Network and Parallel Computing Workshops, pp. 855-862, October, 2007.

[19] M. L. Wong and T. T. Wong, "Implementation of Parallel Genetic Algorithms on Graphics Processing Units", Intelligent and Evolutionary Systems Studies in Computational Intelligence, vol. 187, 2009.

[20] Barcelona, "High Performance Genetic Programming on GPU", in Proc. of 2009 ICAC workshop on Bio-inspired algorithms for distributed systems, pp. 85-94, 2009.

[21] A. S. David, R. J. Randall, J. H. William, P. M. Dwight, Vector, signal and image processing library(VSIPL) 1.3 application programmer's interface, Technical report, Georgia Tech Research Corporation, 2008, http://www.vsipl.org.

[22] H. G. Gene and F. V. Charales, Matrix Computations, Johns Hopkins Universtiy Press, 3rd edtion, 1996

[23] Kerr, D. Campbell and M. Richards, "QR decomposition on GPUs", in Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, vol. 383, pp. 71-78, Washington, D.C. 2009.

[24] V. Volkov and J. W. Dmmel, "Benchmarking GPUs to tune dense linear algebra", IEEE High Performance Computing, Networking, Storage and Analysis, pp. 1-11, 2008.

[25] S. Lahabar, P.J. Narayanan, "Singular Value Decomposition on GPU using CUDA", IEEE Symposium on Parallel & Distributed Processing, pp. 1-10, May, 2009.

[26] S. Mu, X. Y. Zhang, N. R. Zhang, J. X. Lu, Y. Deng and S. Zhang, "IP routing processing with graphic processors", Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 93-98, 2010.

[27] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, et al. "Fast: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs", in Proc. of the 2010 international conference on Management of data, pp. 339-350, 2010.

[28] ADSP-TS101 TigerSHARC 6U cPCI board, http://www.bittware.com/products/hardware/dsp.cfm

[29] Bakhoda, A. Yuan, G. L. Fung, W. W. L. Wong and H. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator", IEEE Symposium on Performance Analysis of Systems and Software, pp. 163-174, Boston MA, April 2009.