General-Purpose vs. GPU: Comparison of Many-Cores on Irregular Workloads

George C. Caragea, Fuat Keceli, Alexandros Tzannes, and Uzi Vishkin
University of Maryland, College Park
{gcaragea,keceli,tzannes,vishkin}@umd.edu

Abstract

XMT[1] is a general-purpose many-core parallel architecture. The foremost design objective for XMT was to meet the highest standards for ease of parallel programming. GPUs, on the other hand, have acquired a strong reputation on performance, sometimes at the expense of ease-of-programming. The current paper presents a performance comparison on diverse workloads between XMT and an NVIDIA CUDA-enabled GPU. Configured with roughly the same amount of chip resources as the GPU, XMT achieves an average speedup of 6.05x on irregular applications, while incurring an average slowdown of 2.07x on regular ones. Namely, XMT comes ahead for significant applications without having to pay a (possibly worthwhile) price for easier programming. This surprising result suggests a yet untapped opportunity: A high-performance easy-to-program general-purpose 1000-core computer.

1 Introduction

Multiple core and multithreaded processors will be the pervasive computing platform of the future. Currently, the two predominating paradigms are: (1) Limited-scale multi-cores that replicate the single-processor model on one die and strive to maintain backwards compatibility. They generally target applications with low degrees of parallelism, programmed to take advantage of local caches and limit expensive inter-core communication. Presently such systems have 2-8 cores, each supporting coarse grained threads and they are not expected to exceed around 20 cores in the foreseeable future. And (2) Many-cores that are not typically confined to traditional architectures and programming models, and use hundreds of lightweight cores in order to provide stronger speedups. GPUs are the the main example, performing best on applications with very high degrees of parallelism; at least 5,000 – 10,000 threads according to [25]. Advances in GPU programming languages (by GPU vendors NVIDIA – CUDA [22], AMD – Brook [5], the upcoming OpenCL standard [20]), and architecture upgrades have led to strong performance demonstrated for a considerable range of software. When all optimizations are applied correctly by the programmer, GPUs provide remarkable speedups for certain types of applications. As of January 2010, the NVIDIA CUDA Zone website [24] lists 198 CUDA reported applications, 28 of which reporting speedups of $100\times$ or more. On the other hand, the programming effort required to extract performance can be quite tedious. The fact that the implementation of basic algorithms on GPUs, such as sorting, merit so many research papers (e.g., [4, 8, 27]) affirms that. Nevertheless, the notable performance benefits led some researchers to regard GPUs as the most promising solution for the pervasive computing platform of the future. The emergence of General-Purpose GPU (GPGPU) communities is perhaps one indication of this belief.

A radically different approach tries to look-up to the serial paradigm in order to understand what made it such a success. In particular, the serial general-purpose paradigm gave "dreamers" of new applications a proper intellectual and business environment that facilitated innovation. We aspire to create a similar environment for future innovators. The main features of the serial paradigm include: a simple abstraction at the heart of the "contract" between programmers and builders, the software spiral (the cyclic process of hardware improvements leading to software improvements, which lead back to hardware improvements and so on), ease-of-programming, and backwards compatibility on existing code and on application programming. The only serial feature that seems generally impossible to provide in a future dominated by pervasive parallel systems is continued performance improvement for serial code. The above discussion motivated the XMT framework (e.g., [29]), a general-purpose manycore architecture. Prior work showed that XMT has the potential to outperform systems such as Intel Core 2 and AMD Opteron processors [6, 26, 34], and require a much lower learning and programming effort [15, 28, 32].

The main contributions of this paper are:

- A meaningful performance comparison of a state-of-the-art GPU to XMT, a general-purpose highly parallel architecture, on a range of irregular applications. We show via simulation that XMT can outperform GPUs on these applications, while not falling behind significantly on regular ones. Fig. 3 summarizes our results.
- Beyond the specific comparison to XMT, the results demonstrate that an easy to program, truly general-purpose architecture can challenge a performance-oriented architecture – a GPU – once applications exceed a specific scope of the latter.

Related Work. Numerous works evaluate and compare

[1] This refers to the XMT architecture developed at the University of Maryland and not the Cray XMT system.

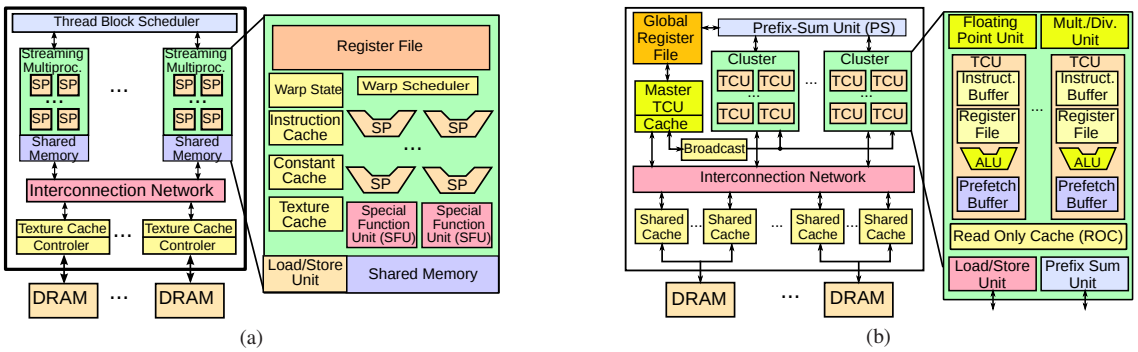


Figure 1: Overview of the compared architectures. (a) The Tesla architecture. (b) The XMT architecture.

different parallel architecture approaches. Some studies focus on a particular application (e.g. [10, 16, 36, 11, 4]), while others aim to run a comparison on heterogeneous benchmark suites (e.g. [9]). In this paper, we focus on two architectures with similar performance goals and means (accelerating single-task completion time through fine grained parallelism), and examine the impact that the design objective of supporting ease of programming has on execution time.

2 The Tesla/CUDA and XMT Frameworks

Tesla/CUDA Framework. In the recent years, the GPU architectures have evolved from purely fixed-function devices to increasingly flexible, massively parallel programmable processors. The CUDA [22, 23] programming environment together with the NVIDIA Tesla [18] architecture is one example of a GPGPU system gaining acceptance in the parallel computing community.

Fig. 1.a depicts an overview of the Tesla architecture. It consists of an array of Streaming Multiprocessors (SMs), connected through an interconnection network to a number of memory controllers and off-chip DRAM modules. Each SM contains a shared register file, shared memory, constant and instruction caches, special function units and several Streaming Processors (SPs) with integer and floating point ALU pipelines. SFUs are 4-wide vector units that can handle complex floating point operations. The CUDA programming and execution model are discussed elsewhere [18].

The CUDA framework provides a relatively familiar environment for developers, which led an impressive number of applications to be ported since its introduction [24]. Nevertheless, a non-trivial development effort is required when optimizing an application in the CUDA model. Some of the considerations that must be addressed in order to get real performance gains follow. *Degree of parallelism:* a minimum of 5,000 - 10,000 of threads need to be in-flight for achieving good hardware utilization and latency hiding. *Thread divergence:* in the CUDA Single Instruction Multiple Threads (SIMT) model, divergent control flow between threads causes serialization, and programmers are encouraged to minimize it. *Shared memory:* no standard cache is included at the SM.[2] Instead, a small user-controlled scratch-pad shared memory is provided. *Memory request coalescing:* better bandwidth utilization is achieved when data layout and memory requests follow a number of temporal and spatial locality guidelines. *Bank conflicts:* concurrent requests to one bank of the shared memory incur serialization, and should be avoided in the code, if possible.

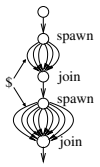
XMT Framework. The primary goal of the eXplicit Multi-Threading (XMT) on-chip general-purpose computer architecture [31, 21] has been improving single-task performance through parallelism. XMT was designed from the ground up to capitalize on the huge on-chip resources becoming available in order to support the formidable body of knowledge, known as Parallel Random Access Model (PRAM) algorithmics, and the latent, though not widespread, familiarity with it. A 64-core FPGA prototype was evaluated in [33, 34, 35].

The XMT architecture, depicted in Fig. 1.b, includes an array of lightweight cores, Thread Control Units (TCUs) and a serial core with its own cache (Master TCU). The architecture includes several clusters of TCUs connected by a high-throughput interconnection network, for example using a mesh-of-trees (MOT) topology [3, 2]; an instruction and data broadcast mechanism; a global register file (GRF); a prefix-sum unit (PS). The first level of cache is shared and partitioned into mutually-exclusive cache modules sharing several off-chip DDR2 DRAM memory channels. The TCU Load-Store unit applies a hashing function on each address to avoid memory hotspots. Cache modules handle concurrent requests and provide buffering and request reordering to achieve better DRAM bandwidth utilization. Within a cluster, a compiler-managed Read-Only Cache (ROC) is used to store constant values across all threads. TCUs include lightweight ALUs, but the more expensive

[2] We do not classify the constant and texture caches as regular caches since they are read-only and use separate address spaces. This changed in the NVIDIA Fermi architecture, which includes an option to use part of the shared memory as an L1 cache. At the time of this writing, it is not clear yet what implications this has on programmability and performance. What limited our choice of a GPU architecture for the comparison in this paper was the availability of third party application code.

```
int A[N],B[N],base=0;
spawn(0,N-1) {
  int inc=1;
  if (A[$]!=0) {
    ps(inc,base);
    B[inc]=A[$];
} }
```

(a)



(b)

Figure 2: (a) XMTC program example: Array Compaction. The non-zero elements of array A are copied into an array B. The order is not necessarily preserved. $ refers to the unique thread identifier. After the execution of the prefix-sum statement `ps(inc,base)`, the `base` variable is increased by `inc` and the `inc` variable gets the original value of `base`, as an **atomic** operation. (b) Execution of a sequence of `spawn` and `join` commands.

Multiply/Divide (MDU) and Floating Point Units (FPU) are shared by all TCUs in a cluster. TCUs also feature prefetch buffers utilized via a compiler optimization to hide memory latencies [7].

The underlying programming model of the XMT framework is arbitrary CRCW (concurrent read/write) reduced-synchrony PRAM-like [29], with serial and parallel execution modes. The *spawn* and *join* instructions specify the beginning and the end of a parallel section that contains an arbitrary number of virtual threads sharing the same code, as shown in Fig. 2. An algorithm designed in the XMT model usually permits each thread to progress at its own speed from its initiating spawn to the terminating join, without ever having to busy-wait for other threads, methodology called "independence of order semantics (IOS)." XMT also includes a hardware implementation of a powerful prefix-sum primitive similar in function to the NYU Ultracomputer Fetch-and-Add [12]; it provides constant, low overhead inter-thread coordination, a key requirement for implementing efficient intra-task parallelism. Fig. 2a illustrates the XMTC programming language, a simple SPMD extension of C. The XMTC compiler is based on the GCC 4.0 suite.

XMT allows concurrent instantiation of as many threads as the number of processors. Threads are efficiently started and distributed using prefix-sum for fast dynamic allocation of work and a dedicated instruction broadcast bus. The high-bandwidth interconnection network and the low-overhead creation of many threads facilitate efficient support of fine-grained parallelism.

Ease-of-programming is a necessary condition for the success of a many-core platform, and it is one of the main objectives of XMT. Indications that XMT is an easy-to-program efficient parallel architecture, include: (i) XMT is *based on a rich algorithmic theory* (PRAM) that provides a robust framework for designing and analyzing algorithms, equivalent to the serial model; (ii) using *ease of teaching* as a benchmark, significant evidence regarding superiority of XMT programming relative to alternative parallel approaches, such as MPI, OpenMP and CUDA, has been established; demonstrations in repeated instances include middle-school and up, by independent education experts [28, 32]; (iii) XMT provides a *programmer's workflow* for deriving efficient programs from PRAM algorithms, and reasoning about their execution time and correctness[30], and (iv) in a semester-long study supported through the DARPA HPCS program, the *development time* of XMT was, not surprisingly, shown to be about half that of MPI under circumstances favoring MPI [15].

Comparison of Architectures. The key issues that affect the design of both architectures, and the main differences between them, are summarized in Table 1.

3 Experimental Evaluation

In this section, we present a head-to-head performance comparison of a simulated XMT chip and an NVIDIA GTX 280 GPU in terms of execution time on a set of benchmarks.

Tested configurations. We needed to determine the power-of-two configuration of XMT whose chip resources are in the same ballpark as the GTX 280, the GPU considered. We base our estimation on the detailed data from the ASIC implementation of the MOT interconnection network [2] and a complete 64-TCU XMT integer-only chip, both fabricated in 90nm IBM technology. Our calculations below show that using the same generation technology as the GTX 280, a 1024-TCU XMT configuration could be fabricated.

A 1024-TCU XMT configuration requires 16 times the cluster and cache modules resources of the 64-TCU XMT ASIC prototype, reported by the design tools as 61mm^2. The area of the MOT interconnection network for the envisioned XMT configuration can be estimated as 64mm^2 in 90nm using the data from [2]. Applying a theoretical area scaling factor of 0.5 from 90nm to 65nm technology feature size we obtain an area estimate of $(61 \times 16 + 64) \times 0.5 = 520\text{mm}^2$. We assume that with the same amount of engineering and optimization effort put behind it as for the GPUs, XMT could support a comparable clock frequency and addition of floating point units (whose count, per Table 2, is around 20% of the GTX 280) without a significant increase in area budget. This is why the XMT clock frequency considered in the comparison is the same as the shader clock frequency (SPs and SFUs) of GTX280, which is 1.3GHz. Note that this estimation does not include the cost of memory controllers. The published die area of GTX280 is 576mm^2 in 65nm technology and approximately 10% of this area is allocated for memory controllers [17]. It is reasonable to assume that the difference in the GPU area and the estimated XMT area, which is also $\approx 10\%$, would account for the addition of the same number of controllers to XMT. We expect that very limited area will be needed for XMT beyond the sum of these components since they comprise a nearly full design.

Table 2 gives a comparative summary of the hardware

	Tesla	XMT
Memory Latency Hiding and Reduction	·Heavy multithreading (requires large register files and state aware scheduler) ·Limited local shared scratchpad memory ·No coherent private caches at SM or SP	·Large globally shared cache ·No coherent private TCU or cluster caches ·Software prefetching
Memory and Cache Bandwidth	·Memory access patterns need to be coordinated by the user for efficiency (request coalescing) ·Scratchpad memories prone to bank conflicts	·Relaxed need for user-coordinated DRAM access due to caches ·Address hashing for avoiding memory module hotspots ·High bandwidth mesh-of-trees interconnect between clusters and caches
Functional Unit (FU) Allocation	·Dedicated FUs for SPs and SFUs ·Less arbitration logic required ·Higher theoretical peak performance	·Heavy FUs (FPU and MDU) are shared through arbitrators ·Lightweight FUs (ALU and branch unit) are allocated per TCU. ALUs do not include multiply/divide functionality
Control Flow and Synchronization	·Single instruction cache and issue per SM for saving resources. Warps execute in lock-step (penalizes diverging branches) ·Efficient local synchronization and communication within blocks. Global communication is expensive ·Switching between serial and parallel modes (i.e. passing control from CPU to GPU) requires off-chip communication	·One instruction cache and program counter per TCU enables independent progress of threads ·Coordination of threads can be performed via constant time prefix-sum. Other forms of thread communication are done over the shared cache ·Dynamic hardware support for fast switch between serial and parallel modes and load balance of virtual threads

Table 1: Implementation differences between XMT and Tesla. FPU and MDU stand for floating-point and multiply/divide units respectively.

	GTX280	XMT-1024
Principal Computational Resources		
Cores	240 SP, 60 SFU	**1024 TCU**
Integer Units	**240 ALU+MDU**	1024 ALU, 64 MDU
Floating Point Units	**240 FPU, 60 SFU**	64 FPU
On-chip Memory		
Registers	**1920KB**	128KB
Prefetch Buffers	–	**32KB**
Regular caches	480KB	**4104KB**
Constant cache	**240KB**	128KB
Texture cache	**480KB**	–

Table 2: Hardware specifications of the GTX280 and the simulated XMT configuration. In each category, the emphasized side marks the more area-intensive implementation.

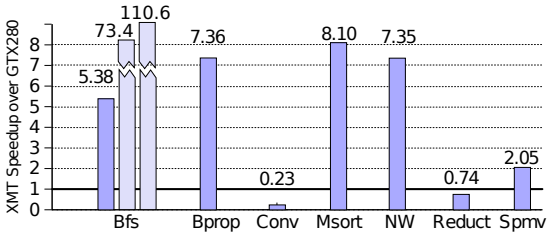


Figure 3: Speedups of the 1024-TCU XMT configuration with respect to GTX280. A value less than 1 denotes slowdown.

specifications of an NVIDIA GTX280 and the simulated XMT configuration. The sharp differences in this table are due to the different architectural design decisions summarized in Table 1. From these calculations, we conclude that overall, the configurations of these very different architectures appear to use roughly the same amount of resources. We also evaluated the performance of a XMT configuration using only 512 TCUs – a very conservative estimation, to account for unforeseen overheads.

Data collection. On the GPU, we compiled and ran CUDA code optimized by others, and collected timing information. On XMT, we compiled and simulated our XMTC implementations on XMTSim, the cycle-accurate simulator of the XMT architecture. XMTSim is modeled after the FPGA implementation, but it can be customized to realistically simulate any configuration, beyond the resource limitations of the FPGA prototype. At this time, off chip buses and DRAM modules are modeled as simple latency components in the simulator. In the 1024-TCU configuration, the latency and bandwidth are set to approximately match the specifications of the GPU. The XMT compiler and simulator are publicly available [1].

Benchmarks. A "general-purpose" architecture should provide good performance on both regular and irregular applications. This guided the selection of benchmarks for this study, as listed in Table 3. We selected benchmarks whose GPU results are published and CUDA source code made available by authors. This ensures that we are using the most optimized code for the CUDA implementation, highly tuned for GPUs. The XMT implementations of the benchmarks were developed by members of the XMT project. The significantly lower number of lines of code of the XMT implementations bring supporting evidence to the ease-of-programming claim.

Note that all our benchmarks use single-precision floating point arithmetic only, to allow for a fair comparison with Tesla. The addition of better support for double precision in upcoming GPUs will not significantly change the relative results, as the same support can be added to XMT using similar additional resources.

Performance Comparison. Figure 3 presents the speedups of all the benchmarks on a 1024-TCU XMT configuration relative to GTX280. Speedups range between $2.05\times$ and $8.10\times$ for highly parallel irregular benchmarks. For one application (BFS), we demonstrate much stronger speedups for limited parallelism, using a dataset with less available parallelism, a synthetic graph with 1M nodes, 3M edges but a diameter of 50,000 Bfs "levels". With this dataset, the average number of active threads per Bfs iteration is 20 (compared to 87.4K

Name	Description	CUDA implementation source	Lines of Code		Dataset	Parallel sectn.		Threads/sectn.	
			CUDA	XMT		CUDA	XMT	CUDA	XMT
Bfs	Breadth-First Search on graphs	Harish and Narayanan [13], Rodinia benchmark suite [9]	290	86	1M nodes, 6M edges	25	12	1M	87.4K
Bprop	Back Propagation machine learning algorithm	Rodinia benchmark suite [9]	960	522	64K nodes	2	65	1.04M	19.4K
Conv	Image convolution kernel with separable filter	NVIDIA CUDA SDK [23]	283	87	1024x512	2	2	131K	512K
Msort	Merge-sort algorithm	Thrust library [14, 25]	966	283	1M keys	82	140	32K	10.7K
NW	Needleman-Wunsch sequence alignment	Rodinia benchmark suite [9]	430	129	2x2048 sequences	255	4192	1.1K	1.1K
Reduct	Parallel reduction (sum)	NVIDIA CUDA SDK [23]	481	59	16M elts.	3	3	5.5K	44K
Spmv	Sparse matrix - vector multiplication.	Bell and Garland [4]	91	34	36Kx36K, 4M non-zero	1	1	30.7K	36K

Table 3: Benchmark properties

Name	MEM	Idle	ALU	FPU	MD	Misc
Spmv	71.8	2.1	6.1	19.1	0.0	0.9
NW	34.7	50.0	6.0	0.0	3.2	6.2
Bfs	94.7	1.2	2.4	0.0	0.0	1.7
Bprop	93.4	1.4	0.6	1.8	1.0	1.9
Msort	63.7	21.1	4.5	3.2	1.0	6.6
Conv	41.1	0.2	14.4	31.5	0.0	12.8
Reduct	71.0	0.9	3.2	23.0	0.0	1.8

Table 4: Percentage of time on XMT spent executing memory instructions (MEM), idling (due to low parallelism), integer arithmetic (ALU), floating-point (FPU), integer multiply-divide (MD) and other.

threads/iteration above). For this input, the XMT implementation exhibited a speedup of 73.4× over [9], and 6.89× when compared to a CUDA Bfs implementation for regular, low degree graphs [19], even when their input processing was not counted. Furthermore, when a 64-TCU XMT configuration was used, the speedup compared to [9] was 110.6×, the better result explained by the lower latencies in the simpler 64-TCU design, with still enough hardware to handle the problem parallelism.

The two regular benchmarks (Conv and Reduct) show slowdown. This is due to the nature of the code, exhibiting regular patterns that the GPUs are optimized to handle, while the XMT abilities to dynamically handle less predictable execution flow go underused. Moreover, Conv on CUDA uses the specialized Tesla multiply-add instruction, while on XMT two instructions are needed.

Table 3 shows the number of parallel sections executed and the average number of threads per parallel section for each benchmark. Table 4 provides the percentage of the execution time spent executing instructions in different categories as reported by the XMT Simulator. To the best of our knowledge, there is no way of gathering such detailed data from the NVIDIA products at this time.

We observed that benchmarks with irregular memory access patterns such as Bfs, Spmv and Msort spend a significant amount of their time in memory operations. We believe that the high amount of time spent by Bprop is due to the amount of memory queuing in this benchmark. Conv is highly regular with lots of data reuse, and spends less than half of its time on memory accesses; however, it performs a non-trivial amount of floating-point computation (more than 50% of the remaining time).

Table 4 shows that in the NW benchmark, a significant amount of time is spent idling by the TCUs. From Table 3, we observe that the number of threads per parallel section is relatively low in this benchmark. In spite of this high idling time, XMT outperforms the GPU by a factor of 7.36x on this benchmark, illustrating the fact that XMT performs well even on code with relatively low amounts of parallelism. The very large number of parallel sections executed for the NW benchmark (required by the lock-step nature of the dynamic programming algorithm) favors XMT and its low-overhead synchronization mechanism, and explains the good speedup.

When using a smaller XMT configuration with only 512 TCUs, we observed that the speedup vs. Tesla for the irregular benchmarks was 4.57x on average, while the slowdown was 3.06x for the regular ones. This shows that such an XMT configuration still outperforms the GPU considered, and given XMT's advantage on ease-of-programming, the main point of this comparison holds.

4 Conclusion

Our work constructively questions what the industry currently offers. In [6, 26, 34] we already compared XMT to existing general-purpose systems. In this paper, we compare XMT, a general-purpose parallel architecture, with a recent NVIDIA GPU programmed using the CUDA framework. We showed that when using an equivalent configuration, XMT outperformed the GPU on all irregular workloads considered. Performance results on regular workloads show that even though GPUs are optimized for these kind of applications, XMT does not fall behind significantly, not an unreasonable price to pay for ease of programming and programmer's productivity.

This paper raises for consideration a promising candidate for the general-purpose pervasive platform of the future, a system consisting of an easy-to-program, highly parallel general-purpose CPU coupled with (some form of) a parallel GPU – a possibility that appears to be underrepresented in current debate. XMT has a big advantage on ease-of-programming, offers compatibility on serial code and rewards even small amount of parallelism with speed-ups over uni-processing, while the GPU could be used for the applications on which it has an advantage.

References

[1] Software release of the explicit multi-threading (xmt) programming environment. http://www.umiacs.umd.edu/users/vishkin/XMT/sw-release.html, August 2008.

[2] BALKAN, A. O., HORAK, M. N., QU, G., AND VISHKIN, U. Layout-accurate design and implementation of a high-throughput interconnection network for single-chip parallel processing. *hoti* (2007), 21–28.

[3] BALKAN, A. O., QU, G., AND VISHKIN, U. A mesh-of-trees interconnection network for single-chip parallel processing. In *ASAP '06: Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 73–80.

[4] BELL, N., AND GARLAND, M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 2009), ACM.

[5] BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph. 23*, 3 (2004), 777–786.

[6] CARAGEA, G. C., SAYBASILI, A. B., WEN, X., AND VISHKIN, U. Brief announcement: performance potential of an easy-to-program pram-on-chip prototype versus state-of-the-art processor. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures* (New York, NY, USA, 2009), ACM, pp. 163–165.

[7] CARAGEA, G. C., TZANNES, A., KECELI, F., BARUA, R., AND VISHKIN, U. Resource-aware compiler prefetching for many-cores. In *Proc. International Symposium on Parallel and Distributed Computing (ISPDC)* (July 2010), IEEE.

[8] CEDERMAN, D., AND TSIGAS, P. On sorting and load balancing on gpus. *SIGARCH Comput. Archit. News 36*, 5 (2008), 11–18.

[9] CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W., LEE, S.-H., AND SKADRON, K. Rodinia: A benchmark suite for heterogeneous computing. In *In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)* (October 2009), IEEE.

[10] CHRISTEN, M., SCHENK, O., MESSMER, P., NEUFELD, E., AND BURKHART, H. Parallel data-locality aware stencil computations on modern micro-architectures. In *23rd IEEE International Parallel and Distributed Processing Symposium* (May 2009).

[11] FALCÃO, G., SILVA, V., AND SOUSA, L. How gpus can outperform asics for fast ldpc decoding. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing* (New York, NY, USA, 2009), ACM, pp. 390–399.

[12] GOTTLIEB, A., GRISHMAN, R., KRUSKAL, C. P., MCAULIFFE, K. P., RUDOLPH, L., AND SNIR, M. The nyu ultracomputer: designing a mimd, shared-memory parallel machine (extended abstract). In *ISCA '82: Proceedings of the 9th annual symposium on Computer Architecture* (Los Alamitos, CA, USA, 1982), IEEE Computer Society Press, pp. 27–42.

[13] HARISH, P., AND NARAYANAN, P. J. Accelerating large graph algorithms on the gpu using cuda. In *In proceedings High Performance Computing - HiPC* (2007), pp. 197–208.

[14] HOBEROCK, J., AND BELL, N. Thrust: A parallel template library, 2009. Version 1.1.

[15] HOCHSTEIN, L., BASILI, V. R., VISHKIN, U., AND GILBERT, J. A pilot study to compare programming effort for two parallel programming models. *Journal of Systems and Software 81*, 11 (2008), 1920 – 1930.

[16] KANG, S., BADER, D. A., AND VUDUC, R. Understanding the design trade-offs among current multicore systems for numerical computations. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 1–12.

[17] KANTER, D. Nvidia's gt200: Inside a parallel processor. physical implementation. http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242&p=11, September 2008.

[18] LINDHOLM, E., NICKOLLS, J., OBERMAN, S., AND MONTRYM, J. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro 28*, 2 (2008), 39–55.

[19] LUO, L., WONG, M., AND MEI HWU, W. An effective gpu implementation of breadth-first search. In *Proc. of Design Automation Conference (DAC)* (2010). To appear.

[20] MUNSHI, A. Opencl specification version 1.0. Tech. rep., Khronos OpenCL Working Group, 2009.

[21] NAISHLOS, D., NUZMAN, J., TSENG, C.-W., AND VISHKIN, U. Towards a first vertical prototyping of an extremely fine-grained parallel programming approach. In *SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures* (New York, NY, USA, 2001), ACM, pp. 93–102.

[22] NICKOLLS, J., BUCK, I., GARLAND, M., AND SKADRON, K. Scalable parallel programming with cuda. *Queue 6*, 2 (2008), 40–53.

[23] NVIDIA. *NVIDIA CUDA SDK 2.3*. NVIDIA Corporation, Santa Clara, California, 2009.

[24] NVIDIA. Cuda zone. http://www.nvidia.com/cuda, 2010.

[25] SATISH, N., HARRIS, M., AND GARLAND, M. Designing efficient sorting algorithms for manycore gpus. In *Proc. 23rd IEEE International Parallel and Distributed Processing Symposium* (May 2009).

[26] SAYBASILI, A. B., TZANNES, A., BROOKS, B. R., AND VISHKIN, U. Highly parallel multi-dimentional fast fourier transform on fine- and coarse-grained many-core approaches. In *PDCS '09: The 21st IASTED International Conference on Parallel and Distributed Computing and Systems* (2009).

[27] SINTORN, E., AND ASSARSSON, U. Fast parallel gpu-sorting using a hybrid algorithm. *J. Parallel Distrib. Comput. 68*, 10 (2008), 1381–1388.

[28] TORBERT, S., VISHKIN, U., TZUR, R., AND ELLISON, D. Is teaching parallel algorithmic thinking to high-school student possible? one teacher's experience. In *Proc. 41st ACM Technical Symposium on Computer Science Education (SIG CSE)* (Milwaukee, WI, March 2010).

[29] VISHKIN, U. Using simple abstraction to guide the reinvention of computing for parallelism. *Communications of the ACM (CACM)* (2010). To appear. Download from http://www.umiacs.umd.edu/users/vishkin/XMT/cacm2010.pdf.

[30] VISHKIN, U., CARAGEA, G. C., AND LEE, B. C. *Handbook of Parallel Computing: Models, Algorithms and Applications*. CRC Press, 2007, ch. Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform.

[31] VISHKIN, U., DASCAL, S., BERKOVICH, E., AND NUZMAN, J. Explicit multi-threading (xmt) bridging models for instruction parallelism (extended abstract). In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures* (New York, NY, USA, 1998), ACM, pp. 140–151.

[32] VISHKIN, U., TZUR, R., ELLISON, D., AND CARAGEA, G. C. Programming for high schools. Keynote,The CS4HS Workshop. Download from http://www.umiacs.umd.edu/~vishkin/XMT/CS4HS_PATfinal.ppt, July 2009.

[33] WEN, X., AND VISHKIN, U. Pram-on-chip: first commitment to silicon. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures* (New York, NY, USA, 2007), ACM Press, pp. 301–302.

[34] WEN, X., AND VISHKIN, U. Fpga-based prototype of a pram-on-chip processor. In *CF '08: Proceedings of the 2008 conference on Computing frontiers* (New York, NY, USA, 2008), ACM, pp. 55–66.

[35] WEN, X., AND VISHKIN, U. The xmt fpga prototype/cycle-accurate-simulator hybrid. In *WARP08: The 3rd Workshop on Architectural Research Prototyping* (Beijing, China, June 2008). In conjunction with ISCA 2008.

[36] WILLIAMS, S., OLIKER, L., VUDUC, R., SHALF, J., YELICK, K., AND DEMMEL, J. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing 35*, 3 (2009), 178 – 194. Revolutionary Technologies for Acceleration of Emerging Petascale Applications.