Power-Efficient Medical Image Processing using PUMA

Ganesh Dasika, Kevin Fan, and Scott Mahlke Advanced Computer Architecture Laboratory University of Michigan - Ann Arbor, MI {gdasika, fank, mahlke}@umich.edu

Abstract-

Graphics processing units (GPUs) are becoming an increasingly popular platform to run applications that require a high computation throughput. They are limited, however, by memory bandwidth and power and, as such, cannot always achieve their full potential. This paper presents the PUMA architecture - a domain-specific accelerator designed specifically for medical imaging applications, but with sufficient generality to make it programmable. The goal is to closely match the performance achieved by GPUs in this domain but at a fraction of the power consumption. The results are quite promising - PUMA achieves upto 2X the performance of a modern GPU architecture and has upto a 54X improved efficiency on a floating-point and memory-intensive MRI reconstruction algorithm.

I. INTRODUCTION

The advent of programmable graphics processing units, or GPUs, for general-purpose computing is one of the major steps taken in computing over the last few years. These GPGPUs which, in the past, have been predominantly used for gaming and advanced image and video editing, are now being used by many developers to accelerate inherently parallel programs in several other domains. Indeed, considerable amounts time and engineering effort are often spent in order to modify programs so that they may run effectively on GPUs.

Several different application domains observe remarkable speedups when using GPUs, including the following [18]:

- 4X to 100X speedup on medical applications, such as biomedical image analysis, 3D reconstruction of tissue structures for large sets of microscopic images and accelerating MRI reconstructions.
- 8X to 260X speedup on electronic design automation, such as power grid analysis and statistical static timing analysis.
- 4X to 327X speedup on physics applications, such as weather prediction and astrophysics.
- 11X to 100X speed up financial applications such as instrument pricing using Monte-Carlo methods.

A. The Advantages of GPUs

GPUs have many appealing hardware features. Firstly, they lend themselves very well to both thread-level and data-level parallelism. Thread-level parallelism (TLP) is exploited by having a large number of independent processing elements (PEs) on the GPU, each with its own set of functional units (FUs) and local storage. Individual threads can quite cleanly be assigned, either statically by the programmer or dynamically by the hardware, to each of these PEs and inter-thread communication is made possible by some form of interconnect fabric or through local storage such as caches. Programs with a large amount of data-level parallelism (DLP) can make use of vector-SIMD units in these PEs which allow a single instruction to perform an operation on several data at the same time. DLP can also be extracted in programs with compute-intensive loops that have little or no interiteration dependencies by executing operations from different iterations within a single SIMD instruction.

Secondly, GDDR RAM and its increasingly fast successors have allowed for GPUs to have access to an immense amount of memory bandwidth. The AMD Radeon HD 4870 - the first GPU to support GDDR5 memory - has a memory bandwidth of up to 115 GB/s.

Above all, GPUs are commodity hardware products commonly available as a part of many desktop and laptop computers. The tools to program them are also easily available; NVIDIA's Compute Unified Device Architecture (CUDA) package, for example, is free to download from their website [15]. CUDA is a general purpose parallel computing architecture which consists of the CUDA instruction set and the compute engine in the GPU. It provides a small set of extensions to the C programming language, which enables straightforward implementation of parallel algorithms on the GPU. CUDA also supports scheduling the computation between CPU and GPU, such that serial portions of applications run on the CPU and parallel portions are mapped to the GPU.

Individual cores in Intel's up-and-coming Larrabee processor implement the ubiquitous x86 ISA [23], allowing users to use a host of already-existing development tools to port their applications to it. Server products like Tesla [17] with even more compute power are also available.

B. The Disadvantages of GPUs

One of the main bottlenecks in applications running on GPUs is the gap between their computational ability and their memory bandwidth. While the absolute memory bandwidth available to GPUs is quite high, it is not growing at the same rate as their theoretical peak performance. Currently, even one of the latest generation of NVIDIA graphics processors, the GeForce GTX 280 [16] can only transfer up to 142 GB/s of data despite having a peak performance of 933 GFLOPS [2]. This works out to approximately 0.15 bytes of data, on average, per floating-point operation.

Further, the total power consumption of these GPUs, is enormous. The GeForce GTX 280, for example, while having a tremendous amount of compute ability, consumes 236W of power resulting in a very low MIPSper-mW power-efficiency ratio at peak performance. Inside GPU cores, the main power-consuming components are general storage elements such as register files and scratchpad memories. This is not necessarily a drawback with GPUs, specifically, but rather with most general-purpose programming devices. The interconnect required to access random elements in register files is quite power-hungry, and the more read and write ports a register file has, the higher its power consumption. Consequently, register files that feed vector-SIMD FUs and must output several elements at once consume significant amounts of power.

Though power is not necessarily a significant drawback for users interested in using these devices for video game graphics acceleration, it is not a desirable choice when on a limited power source, in an embedded system or in a high-temperature environment. To address these disadvantages, many designers turn to customized and domainspecific hardware.

C. The Quest for Programmable and Specialized Hardware

A wide range of architectures, in addition to GPUs, have been designed before to address the problem of providing high performance computation efficiently. These solutions maintain or sacrifice programmability to various degrees depending on the domain they target. Figure 1 shows the performance (on the y-axis) and programmability (on the x-axis) expectations from various architecture styles. The numbers next to each of the ovals shows the approximate performance-power ratio offered by each of these solutions.

General purpose processors (GPPs) which fall on the lower right corner of the figure, are highly programmable solutions but are limited in terms of the peak performance they can achieve. Further, structures like instruction decoders and caches that are needed to support programmability consume energy. This results in a very low computational efficiency of about 1 MIPS-per-mW, for example, for the Intel Pentium-M processor.



Fig. 1: Comparison of peak performance, power efficiency, and programmability of different architecture design styles.

On the other end of the spectrum are Application-specific Integrated Circuits (ASICs). ASICs are custom-designed specifically for a particular problem, without extraneous hardware structures. Thus, ASICs have a high computational density with hardwired control, resulting in high computation efficiency up to 1,000 to 10,000 times more than that of GPPs. The space between these two extremes is populated by different solutions that have varying degrees of programmability.

Application specific instruction-set processors (ASIPs) are processors with custom extensions for a particular application or applicationdomain. They can be quite efficient when running the applications for which they are designed, and they are also capable of running any other application, though with reduced efficiency. Examples include processors from Tensilica and ARC, transport triggered architectures [3] and custom-fit processors [9].

Domain loop accelerators are designed to execute computation intensive loops present in media and signal processing domains. Their design is close to that of VLIW processors, but with a much higher number of function units, and consequently, a higher peak performance. Very long instruction words in a control memory direct all FUs every cycle. However, domain loop accelerators (LAs) have less flexibility than GPPs because only highly computationally-intensive loops map well to them. Some examples of architectures in this design space are RSVP [1] and CGRAs [14].

Coarse-grain adaptable architectures have coarser-grain building blocks compared to FPGAs, but, like FPGAs, still maintain bit-level reconfigurability. The coarser reconfiguration granularity improves the computation efficiency of these solutions. However, non-standard tools are needed to map computations onto them and their success have been limited to the multimedia domain. PipeRench [10], RaPiD [6] are some examples of coarse-grain adaptable architectures.

D. Programmable Loop Accelerators

The programmable solutions shown in Figure 1 are all "universally" programmable, allowing any loop to be mapped on to them, although at varying degrees of efficiency. There is a wide gap between the efficiency that can be achieved by ASICs and the efficiency that can be achieved by these programmable solutions. There are, for example, instances where there is a narrow requirement of flexibility. Using any of these above solutions is overkill for these instances as these solutions sacrifice too much efficiency for the needed flexibility. Further, most of the middle-ground solutions listed above do not offer any support for fast floating-point computation, which limits the number of applications that they can be used for.

This work advocates an open area in the design space where a non-trivial amount of programmability is provided in terms of intraprocessor communication, functionality and storage, but the application

Benchmark	#instrs	%FP	Data Req'd	
			\overline{instr}	
MRI.FH	38	42	0.95	
MRI.Q	34	35	1.06	
CT.segment	26	42	1.38	
CT.laplace	20	30	1.20	
CT.gauss	22	32	1.09	

TABLE I: Medical application characteristics

and domain-specific design, as a whole, resembles an ASIC more than a processor. The design point is labeled Programmable Loop Accelerator, or PLA (not to be confused with programmable logic array). The specifics of the PLA are described in Section III-A.

II. TARGETING MEDICAL APPLICATIONS

Medical imaging devices are generally large, bulky and expensive machines that have very limited portability and consume large amounts of power. There is an increasing focus on reducing the power of these medical imaging devices [20]. In order to address this issue, this work focuses on principle components of Computed Tomography (CT) and Magnetic Resonance Imaging (MRI) image processing and reconstruction.

A CT scan involves capturing a composite image from a series of X-Ray images taken from various angles around a subject. It produces a very large amount of data that can be manipulated using a variety of techniques to best arrive at a diagnosis. Oftentimes, this involves separating different layers of the captured image based on their radio-densities. A common way of accomplishing this is by using a well-known image-processing algorithm known as "image segmentation".

In essence, image segmentation allows one to partition a given image into multiple regions based on any of a number of different criteria such as edges, colors, textures, etc. Being able to partition images in this manner allows for studying of isolated sections of the image rather than of all the information that was captured.

The segmentation algorithm used in this work has three main floatingpoint-intensive components: Graph segmenting (CT.segment), Laplacian filtering (CT.laplace) and Gaussian convolution (CT.gauss).

Laplacian filtering highlights portions of the image that exhibit a rapid change of intensity and is used in the segmentation algorithm for edge detection. Gaussian convolution is used to smooth textures in an image to allow for better partitioning of the image into different regions.

An MRI scan, instead of using X-Rays, uses a strong magnetic and radio frequency fields to align, and alter the alignment of, hydrogen atoms in the body. The hydrogen atoms then produce a rotating magnetic field that can be detected by the MRI scanner and converted to an image. The main computational component of reconstructing an MRI image is calculating the value of two different vectors, known here as MRI.FH and MRI.Q, respectively (explained in more detail in [13], [24]).

Table I shows some characteristics of the benchmarks in consideration. All of these benchmarks are floating-point-intensive and require large amounts of data for the computation they perform, especially when compared to the 0.15 bytes/instruction supported by the GTX 280 GPU mentioned earlier. The main loops in these benchmarks are "do-all" loops - there are no inter-iteration dependences.

Prior work in this field has predominantly focused on using commercial products to accelerate medical imaging. For instance, in [11], the authors port "large-scale, biomedical image analysis" applications to multi-core CPUs and GPUs, and compare different implementation strategies with each other. In [21], the authors study image registration and segmentation and accelerate those applications by using CUDA on a GPU. In [24], the authors use both the hardware parallelism and the special function units available on an NVIDIA GPU to dramatically improve the performance of an advanced MRI reconstruction algorithm. There are several other such examples of novel work in this field.

In contrast with such research, this work focuses on designing a new, highly efficient, microarchitecture and architecture with the specific hardware requirements of medical imaging in consideration.



Fig. 2: PUMA. Each tile comprises of a programmable loop accelerator (template pictured) and the control and data memories required for its operation. On-chip routers transfer data between each tile and the external interface.

III. PUMA

PUMA, *P*arallel *micro*-architecture for *M*edical *A*pplications, is a tiled architecture as shown in Figure 2. It is specifically designed to maximize power-efficiency when executing medical imaging applications while still retaining full programmability. Each tile in PUMA is an instance of a specialized PLA - a generalized loop accelerator. The PLA tiles are connected to their neighboring tiles and to the external interface through a high-bandwidth mesh of on-chip routers.

A. Background

Figure 3 shows the hardware schema for the single-function loop accelerator [7], [5]. The LA is designed to efficiently execute a modulo scheduled loop [19] in hardware. The length of the schedule, and the corresponding run-time of the loop, are determined by the *initiation interval* (II) - the number of cycles between the beginnings of successive iterations of the loop. Thus, a lower II corresponds to a shorter schedule and increased performance. The modulo schedule contains a *kernel* that repeats every II cycles and may include operations from multiple loop iterations.

The LA is designed to exploit the high degree of parallelism available in modulo scheduled loops with a large number of function units (FUs). Each FU performs a specific set of functions that is tailored for the particular loop. Each FU writes to a dedicated shift register file (SRF); in each cycle, the contents of the registers shift downwards to the next register. Point-to-point wires from the registers to FU inputs allow data transfer from producers directly to consumers. Multiple registers may be



Fig. 3: Template for single-function loop accelerator.

connected to each FU input; a multiplexer (MUX) is used to select the appropriate one. Since the operations executing in a modulo scheduled loop are periodic, the selector for this MUX is essentially a modulo counter. In addition, a central register file (CRF) holds static live-in register values that cannot be stored in the SRFs.

The schema described is a template that is customized for the particular loop being accelerated. The number, types, and widths of the FUs, the widths and depths of the SRFs, and the connections from the SRFs to the FUs are all determined from the loop. During synthesis, the loop is first modulo scheduled to meet a given performance requirement, and then the details of the LA datapath are determined from the communication patterns in the scheduled loop.

The control path for the single-function LA consists of a finite state machine with II states corresponding to each of time slots in the kernel of the modulo schedule. In each state, control signals direct the execution of FUs (for FUs capable of multiple operations) and control the MUXes at the FU inputs.

Finally, a Verilog HDL realization of the accelerator is generated by emitting modules with pre-defined behavioral Verilog descriptions that correspond to the datapath elements. A simulation environment is used to verify that the Verilog properly implements the loop. Finally, gatelevel synthesis, placement, routing, power analysis and post-synthesis verification are performed on the design.

B. PUMA Architecture

1) Baseline PLA Design: A PLA is generalized loop accelerator, created by modifying the template datapath shown in Figure 3. A generic datapath template for the PLA is illustrated on the right side of Figure 2 The accelerator is designed for a specific loop at a specific throughput, but contains a more general datapath than the single-function LA to allow for different loops to be mapped onto the hardware [8]. These generalizations provide the LA with flexibility in functionality, storage, control and communication.

To provide functionality, simple modifications were made to FUs in order for them to support more operations; adders (both integer and floating-point) are generalized to adder/subtracter units, left-shift units are generalized to left/right rotators, every FU can execute an identity operation to act like a move instruction, etc. Even load-store units can be generalized to integer adder/subtracter units if they already had the functionality to compute indirect addresses. Further, the input-muxes to FUs are redesigned to allow for operand-swapping as well.

The SRFs used in the LA have limited addressability and fixed lifetimes for variables. To overcome these constraints and provide more generality, these SRFs are replaced with rotating-register files (RRs).

To improve controllability, the LA's finite state machine is replaced with a control memory, the contents of which can be changed based on the loop that needs to be executed. Further, numerical constants which were hard-coded in the LA are instead stored in a literal register file.

To generalize communication, the PLA has a bus (in addition to the point-to-point connections) that connects all the RRs to all the FUs. To reduce the hardware cost of having a potentially long bus, its width is limited to one operand, but has a predictable latency of one cycle.

Maximize:

$$\sum_{i \in T_{\alpha}} \sum_{j \in T_{\beta}} C_{ij} \qquad \forall \alpha \forall \beta : \alpha \neq \beta$$

Subject to:

(1)	$\sum_{i=0}^{\#FUs} X_{ij} = 5$	$i \in [0, \#FUs)$
(2)	$\tilde{X}_{ii} = 1$	$i \in [0, \#FUs)$
(3)	$C_{ii} = 1$	$i \in [0, \#FUs)$
(4)	$X_{ij} = X_{ji}$	$i, j \in [0, \#FUs)$
(5)	$C_{ij} = C_{ji}$	$i, j \in [0, \#FUs)$
(6)	$C_{ij} \leq X_{ij} + I_{ij}$	$i, j \in [0, \#FUs)$

Fig. 4: ILP formulation for FU arrangement on the PUMA ring

2) PUMA PLA: The PLA bus is not always a viable solution. One main disadvantage with the bus is that it is not a scalable solution for larger PLAs with many FUs. Further, the bus only carries a single operand per cycle, limiting the amount of programmability available in the PLA and the sequences of opcodes that can be executed in parallel.

To overcome these limitations, the intra-PLA communication fabric in PUMA is changed to a ring. A ring allows for as many operands to be transferred as there are connections to FUs. It does have its limitations, however. The assumed single-cycle latency to transfer data between two arbitrary points in the PLA using the bus is no longer valid, as it takes one cycle to transfer an operand from one ring connection (or ring stop) to another. Also, the longer the distance an operand needs to travel on the ring, the more FUs that have to execute move instructions to propagate the operand along at each ring stop. These added instructions can potentially increase the loop's schedule length and reduce the accelerator's performance.

In PUMA, the ring architecture actually consists of six rings (three sets of two rings going in opposite directions). The first set of rings has a Bus/FU connector (or ring-stop) at every single FU. The second set of rings has a ring-stop at all the odd-numbered FUs, and the third set of rings has a ring-stop at all the even-numbered FUs. This effectively connects an FU/RF pair to its two neighbors and also to its neighbors' neighbors; i.e. every FU can communicate with itself or with other FUs one or two positions on either side of it on the ring. With this configuration, the number of cycles required to transmit data between any two arbitrary FUs is no more than $\left\lceil \frac{\#FUs}{4} \right\rceil$, and regardless of the ordering of FUs on the ring, every possible producer-consumer pairing can be executed, provided sufficient time.

In order to best maintain generality, we chose to arrange the FUs along the ring to allow maximum connectivity and to distribute the various types of FUs as evenly as possible. This was done by formulating the problem as an integer linear program (ILP) as shown in Figure 4.

In the objective function, T_{α} and T_{β} are two different sets of FUs, each set having all and only the FUs of a particular type. The subscripts *i* and *j* are FU IDs and C_{ij} is a binary variable that is 1 if a connection exists between FU *i* and FU *j*. Essentially, this objective function aims to maximize the number of connections between different types of FUs, subject to the following constraints:

- In constraint set (1), X_{ij} is a binary variable that is 1 if FU *i* is "positioned" adjacent to FU *j*, implying that they are connected by the ring. Every FU, therefore, is "positioned" next to 5 other FUs: itself, its two neighbors and the two additional FUs neighboring its neighbors.
- Constraint sets (2) and (3) specify that every FU is "positioned" next to and connected to itself.
- Constraint sets (4) and (5) specify that all added connections are bidirectional.
- In constraint set (6), I_{ij} is a binary number that is 1 if a connection between FU *i* and FU *j* has already been inserted by the synthesis tool. This constraint enforces the rule that a connection between FU *i* and FU *j* can only exist if they are either "positioned" next to each other or are already connected.
- A 7th set of constraints was initially used which specified that

Benchmark	Peak Perf.	Peak Perf.	B/W	#Tiles
	$\frac{GFLOps}{sec}$	$\frac{GIOps}{sec}$	$\frac{GB}{sec}$	
MRI.FH	7.2	5.4	16.2	9
MRI.Q	5.4	5.4	16.2	9
CT.segment	4.95	2.25	16.2	9
CT.laplace	2.7	3.15	10.8	14
CT.gauss	3.15	3.6	10.8	14

TABLE II: Characteristics of the individual accelerators for each benchmark.

there must always be a path between any two FUs with exactly $\left\lceil \frac{\#FUs}{4} \right\rceil$ connections between them This constraint was used to prevent insular sets of 5 FUs or sets of 5 FUs connected linearly rather than in a ring (i.e. without a direct connection between the two ends). While this problem might occur in theory, the pre-existing connections put in place by the synthesis system prevent it from happening in practice and these constraints were removed to reduce the size of the ILP.

Once the optimal solution is obtained, the values of the X_{ij} variables provide a unique ring arrangement.

C. PUMA System Architecture

Tiled architectures have been used in several other projects, such as Raw [25], TRIPS [22], MorphoSys [12], Merrimac [4], etc. Such an architecture was chosen for PUMA (as shown in Figure 2) for a few different reasons. The replication of identical tiles means that the application need not be restricted to run on only a few specific portions of the processor, making compilation for PUMA easier. This is especially useful if the processor is used to execute a stream-like application. For example, in the CT scan benchmarks used here, image segmentation can be executed on one part of the image on some tiles, transfer the resultant data to other tiles for filtering, and perform segmentation on a different part of the image. Further, replication of a single PLA design simplifies the top-level system design and makes testing and verification easier.

The programming and execution model of PUMA closely follows that of modern-day general-purpose GPU processors. Like GPGPUs, PUMA is intended to be mainly used to accelerate compute-intensive, highly-parallel loops, but is able to execute all the other sections of the program as well, albeit at reduced performance.

PUMA is currently envisioned to be in one of two forms: either a discrete core on a PCI-Express (or similar) card external to the main processor core (as pictured in Figure 2), or on the same die as the main processor, connected either through memory or via an ultra-high bandwidth, on-chip bus. For the purposes of this work, we have assumed the former model, for more fair, direct comparisons to the current state of the art GPGPUs.

IV. EXPERIMENTS AND RESULTS

A. Setup

All the PLAs in this work were synthesized for (and run at) a frequency of 450 MHz. The logic synthesis was done using Synopsys Design Compiler 2006-06 and Synopsys Physical Compiler 2006-06, targeting a 65nm process technology with a nominal supply voltage of 0.9 Volts. Energy numbers were obtained using Synopsys PrimeTime-PX 2006-12. For the purposes of this study, we assume that a peak memory bandwidth of 142 GB/s is available to each PUMA system. This is the same amount of bandwidth afforded to the NVIDIA GTX 280 processor.

B. PLA Characteristics

PUMA systems were built using PLAs for each of the five benchmarks in considerations (five systems, each composed entirely of multiple tiles of a single type of PLA). Table II shows various characteristics of these accelerators. The "Peak Perf." columns show



Fig. 5: Normalized performance of benchmarks on LA and PUMA PLA designed for MRI.FH



Fig. 6: Normalized $\frac{Performance}{Power}$ efficiency of benchmarks relative to MRI.FH

the throughput when executing floating-point operations and integer operations, respectively, in billions of operations per second. The next column shows the minimum bandwidth required by each application to prevent starvation. Finally, the last column shows the total number of tiles of each PLA that would be present in a PUMA system. The number of tiles was chosen to prevent data starvation, to make the most efficient use of the resources available. For example, the number of tiles in a system with MRI.FH tiles is $\left\lceil \frac{142}{16.2} \right\rceil$, or 9. Figure 5 shows the normalized performance difference between

Figure 5 shows the normalized performance difference between the non-generalized and generalized loop accelerators across various benchmarks, to illustrate the effects of the modifications made to the baseline accelerator to increase programmability. Each of the different benchmarks were compiled for the MRI.FH accelerator.

The left column for each benchmark shows its normalized performance. The benchmarks MRI.Q, CT.laplace and CT.gauss suffered a 50% reduction in performance; i.e. their II values had to be doubled, from 1 to 2, in order for them to execute on the baseline loop-accelerator. The benchmark CT.segment could not be compiled for the MRI.FH accelerator at all.

For each benchmark, the column on the right shows the achieved performance on the generalized accelerator, with the hardware modifications specified in section III-B1. As shown, these modifications allowed all the benchmarks to run at full performance, at minimum II.

Figure 6 shows a graph similar to that in Figure 5, but shows the normalized efficiency in terms of the accelerator's performanceto-power ratio. Due to the increase in overall performance provided by the generalizations, the benchmarks MRI.Q, CT.laplace and CT.gauss had a significant increase in efficiency despite the power overhead of the additions. The MRI.FH benchmark, however, which would not experience any improved performance from the generalizations loses efficiency due to the increase in the accelerator's power consumption. On average, the generalizations increased the accelerator's efficiency increased by approximately 40%.

C. System Characteristics

We evaluated five different PUMA system designs, one for each PLA design. Each system had a different number of tiles, based on the bandwidth requirement of each benchmark as specified in Table II.

Figure 7 shows the total performance offered by the PUMA systems designed around each of the different PLAs, measured in thousands of MIPS. For each benchmark, the column on the left shows the peak



Fig. 7: Overall system performance for each of the PUMA systems



Fig. 8: Run-time of benchmarks running on different PLAs, normalized to that of the native benchmark

raw performance available to all applications. The column on the right shows what the bandwidth-limited performance is while each system is running the benchmark that it was designed for. These theoretical and realized performances are quite close, differing on average by less than 4%.

Figure 8 shows a set of columns for each benchmark, where each column indicates the normalized run-time of the benchmark on different PLAs. These values are normalized to the run-time of the benchmark on a PLA designed for it. All of the benchmarks could be scheduled with an II of 1. Therefore, there are often considerable reductions in run-time when the smaller benchmarks are executed on PLAs designed for the larger benchmarks. The most dramatic example is the difference in the run-times of the CT.gauss benchmark on the CT.laplace and MRI.FH systems. The opposite, of course, also holds: the larger benchmarks suffer a significant increase in run-time when executing on PLAs designed for smaller benchmarks. Of note is the difference in the run-times of the MRI.FH benchmark on the CT.laplace and MRI.FH systems.

Figure 9 shows a similar graph to that in Figure 8, but showing the average energy consumed per iteration by each benchmark while running on PLAs designed for other benchmarks. The energy consumption was primarily determined by the size of each benchmark, with the two MRI benchmarks consuming the most regardless of which PLA they ran on.

The most important thing to note on this graph is that the most energy-efficient system is the one designed for MRI.Q. The main reason for this is that of the five benchmarks in consideration, it is the one that is closest to being the "average benchmark". This is clear from the data presented in Table II. Its data:compute ratio is quite close to the average among the benchmarks providing a good balance between the more compute-intensive benchmarks and the more data-intensive benchmarks. Its integer and floating-point throughput are identical, providing a balance between the more floating-point-intensive benchmarks and the more integer-intensive benchmarks.

D. Commodity GPGPU Comparison

While other architectures may certainly be used for this domain, GPGPUs are the solutions that are currently in use in many medical imaging applications and, therefore, the most suitable comparison point. For this reason, we assessed the performance and efficiency of five NVIDIA GPUs.



Fig. 9: Average energy consumed (per iteration) by each benchmark while running on PUMA systems designed around different PLAs



Fig. 10: Achieved performance of the MRI.FH benchmark (in trillions of operations) on the MRI.FH PUMA system and on various NVIDIA GPUs based on the GT200 architecture

Figure 10 shows the result of direct performance comparisons between an MRI.FH PUMA system and the GPUs in consideration. The column on the left shows the total compute capability of each of the processors. The column on the right shows the realized performance while executing the MRI.FH benchmark, accounting for bandwidth restrictions. PUMA achieves a very small fraction of the peak performance offered by the GPUs, between 8.6% of the dual-GPU GTX 295 and 21.8% of the GTS 250.

This gap changes dramatically, however, when accounting for the bandwidth-intensive nature of the application in question. PUMA delivers between 63% (on the dual-GPU GTX 295) and 2X the performance (on the GTS 250) of the GPUs.

The case for PUMA is further underscored by examining the GPUs' power efficiency, as shown in Figure 11. This graph shows how many times more efficient, in terms of number of operations per Watt, PUMA systems are relative to the GPUs in consideration. These values range from 20X, for the most complex benchmark running on the most efficient GPU, to 54X, for the least complex benchmark running on the least efficient GPU.

V. CONCLUSION

The PUMA architecture is a power-efficient accelerator system designed specifically for efficient medical image reconstruction. It consists of tiles of programmable loop accelerators - ASICs with added hardware to support general-purpose computing - designed around the computation requirements of the image reconstruction domain. As applications in this domain are normally executed on very high-performance GPGPUs, the latest NVIDIA GPU architecture was used to gauge the performance and efficiency of PUMA. The results are very encouraging - PUMA achieves up to 2 times the performance of a modern GPU architecture and has up to 54 times the power efficiency.

ACKNOWLEDGEMENTS

We thank the anonymous referees who provided excellent feedback and Mojtaba Mehrara for his help in writing this paper. We would also like to think Prof. Jeffrey Fessler for providing valuable insight into the medical imaging domain. This research was supported by ARM Ltd. and by the National Science Foundation grant CCF-0347411.





REFERENCES

- [1] S. Ciricescu et al. The reconfigurable streaming vector processor (RSVP). In Proc. of the 36th Annual International Symposium on Microarchitecture, pages 141-150, 2003.
- [2] CNET. The Gizmo Report: NVIDIA's GeForce GTX 280 GPU introduc-
- tion, 2008. http://news.cnet.com/8301-13512_3-9969234-23.html.
 [3] H. Corporaal. TTAs: Missing the ILP complexity wall. *Journal of System Architecture*, 45(1):949–973, 1999.
- [4] W. Dally et al. Merrimac: Supercomputing with streams. In Proceedings of
- The 2003 ACM/IEEE conference on Supercomputing, pages 35–42, 2003.
 G. Dasika, S. Das, K. Fan, S. Mahlke, and D. Bull. DVFS in loop accelerators using BLADES. In Proc. of the 45th Design Automation Conference, pages 894-897, June 2008.
- C. Ebeling et al. Mapping applications to the RaPiD configurable architec-ture. In *Proc. of the 5th IEEE Symposium on Field-Programmable Custom* [6] Computing Machines, pages 106–115, Apr. 1997. [7] K. Fan et al. Cost sensitive modulo scheduling in a loop accelerator
- synthesis system. In Proc. of the 38th Annual International Symposium on Microarchitecture, pages 219–230, Nov. 2005.
- K. Fan et al. Modulo scheduling for highly customized datapaths to increase hardware reusability. In Proc. of the 2008 International Symposium on Code Generation and Optimization, pages 124-133, Apr. 2008.
- [9] J. A. Fisher et al. Custom-fit processors: Letting applications define architectures. In Proc. of the 29th Annual International Symposium on Microarchitecture, pages 324–335, Dec. 1996.
- [10] S. Goldstein et al. PipeRench: A coprocessor for streaming multimedia acceleration. In Proc. of the 26th Annual International Symposium on Computer Architecture, pages 28-39, June 1999
- [11] T. D. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon. Biomedical image analysis on a cooperative cluster of GPUs and multicores. In Proc. of the 2008 International Conference on Supercomputing, pages 15-25, 2008.
- [12] G. Lu et al. The MorphoSys parallel reconfigurable system. In Proc. of the 5th International Euro-Par Conference, pages 727–734, 1999.
- [13] A. Mahesri et al. Tradeoffs in designing accelerator architectures for visual computing. In Proc. of the 41st Annual International Symposium on Microarchitecture, pages 164-175, Nov. 2008.
- [14] B. Mei et al. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In Proc. of the 2003 Design, Automation and Test in Europe, pages 296–301, Mar. 2003.
- [15] Nvidia. CUDA Programming Guide, 2007. June http://developer.download.nvidia.com/compute/cuda.
- [16] NVIDIA. GeForce GTX 280, 2008. http://www.nvidia.com/object/ product_geforce_gtx_280_us.html.
- NVIDIA [17] NVIDIA. Tesla S1070. 2008.http://www.nvidia.com/object/product_tesla_s1070_us.html.
- [18] Nvidia. Cuda Zone, 2009. http://www.nvidia.com/object/cuda_home.html. [19] B. R. Rau. Iterative modulo scheduling: An algorithm for software
- pipelining loops. In Proc. of the 27th Annual International Symposium on Microarchitecture, pages 63-74, Nov. 1994.
- Μ. [20] Т Review. Cheap, Portable MRL 2006 http://www.technologyreview.com/computing/17499/?a=f.
- [21] A. Ruiz, M. Ujaldon, L. Cooper, and K. Huang. Non-rigid registration for large sets of microscopic images on graphics processors. Springer Journal of Signal Processing, May 2008. [22] K. Sankaralingam et al. Exploiting ILP, TLP, and DLP using polymorphism
- in the TRIPS architecture. In Proc. of the 30th Annual International Symposium on Computer Architecture, pages 422-433, June 2003.
- [23] L. Seiler et al. Larrabee: a many-core x86 architecture for visual computing. ACM Transactions on Graphics, 27(3):1-15, 2008.
- S. S. Stone et al. Accelerating advanced MRI reconstructions on GPUs. In [24] 2008 Symposium on Computing Frontiers, pages 261–272, 2008.
- M. Taylor et al. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proc. of the 31st Annual* [25] International Symposium on Computer Architecture, pages 2-13, June 2004.