Dissecting Cyclops: A Detailed Analysis of a Multithreaded Architecture

George Almási, Călin Caşcaval, José G. Castaños, Monty Denneau,
Derek Lieber, José E. Moreira, Henry S. Warren, Jr.
{gheorghe,cascaval,castanos,denneau,lieber,jmoreira,hankw}
@us.ibm.com
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598-0218

Abstract

Multiprocessor systems-on-a-chip offer a structured approach to managing complexity in chip design. Cyclops is a new family of multithreaded architectures which integrates processing logic, main memory and communications hardware on a single chip. Its simple, hierarchical design allows the hardware architect to manage a large number of components to meet the design constraints in terms of performance, power or application domain.

This paper evaluates several alternative Cyclops designs with different relative costs and trade-offs. We compare the performance of several scientific kernels running on different configurations of this architecture. We show that by increasing the number of threads sharing a floating point unit we can hide fairly high cache and memory latencies. We prove that we can reach the theoretical peak performance of the chip and we identify the optimal balance of components for each application. We demonstrate that the design is well adapted to solve problems that are difficult to optimize. For example, we show that sparse matrix vector multiplication obtains 16 GFlops out of 32 GFlops of peak performance.

1. Introduction

Multiprocessor systems-on-a-chip typically consist of multiple instances of different components: (i) functional units; (ii) memory (including cache and main memory); and (iii) interconnection network. Design choices include both the relative and absolute numbers for components, their particular features, and their placement with respect to each other. To make appropriate design decisions, the architect must be able to quantify the impact of these choices, both on cost (area, power, circuit complexity) and performance.

In this paper, we describe our approach to the problem of managing the many design alternatives for one particular multiprocessor system: the Cyclops chip, which is a new family of multithreaded architectures being developed at the IBM T. J. Watson Research Center. A single Cyclops chip consists of a large number (typically hundreds) of simple thread execution units, each one simultaneously executing an independent stream of instructions. The performance of each individual thread is not particularly high, but the aggregate chip performance is much better than any conventional design with an equivalent number of transistors. Cyclops also uses a processor-in-memory (PIM) design where main memory and processing logic are combined into a single piece of silicon. Large, scalable systems can be built with a cellular approach using Cyclops as a building block, with the cells interconnected in a regular pattern through communication links provided in each chip.

The study of single chip performance performed in this paper provides valuable information to the hardware architects to determine the low-level characteristics of the design and provides a starting point for the future evaluation of multichip Cyclops systems. We begin with a reference design for the Cyclops architecture, representing one particular alternative in terms of various design parameters. We then conduct a systematic exploration of the design space by performing simulation-based sensitivity analysis along each dimension on a suite of scientific kernels. The goal of this design space exploration is not simply to show that one design point is better than another, but to develop an insight into how the numbers and features of components impact the behavior of the chip. Armed with this insight, and with the knowledge of the cost of the various components, the architect can them make design decisions that result in a chip optimized for an application domain.

The rest of this paper is organized as follows. Section 2 provides an overview of the Cyclops architecture and summarizes the parameters used in our initial configuration.

Section 3 presents a short overview of the Cyclops system software and of our architecturally accurate software simulation environment. Section 4 explains the scientific kernels used in this study. The experimental results obtained from simulations are presented in Section 5. Section 6 compares Cyclops against other system-on-a-chip projects and we conclude in Section 7.

2. The Cyclops Architecture

Cyclops is a shared address space multithreaded architecture with a hierarchical and cellular organization, as shown in Figure 1. The main premise of our architecture is the integration of memory, interconnection logic, and a large number of simple concurrent threads in a single chip. Rather than hiding latency through out-of-order or speculative execution, Cyclops tolerates it through massive parallelism. In this design each thread unit is simple; expensive resources, such as floating-point units and caches, are shared between different threads. The integration of memory and logic in the same chip results in a flat, high bandwidth and low latency memory hierarchy.

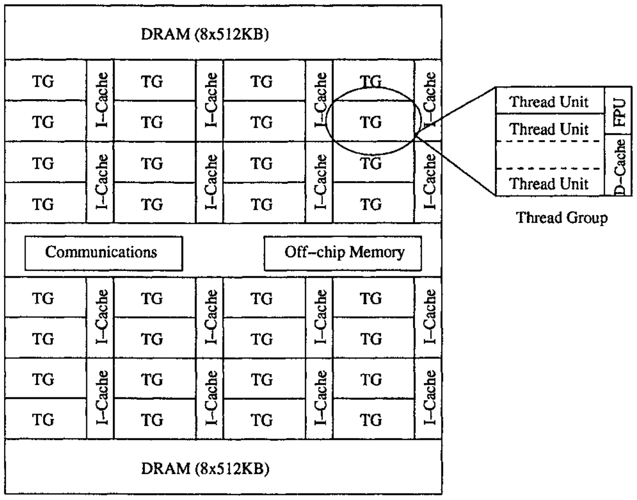


Figure 1. Cyclops chip block diagram.

Thread units are at the base of the Cyclops hierarchy. Each thread unit consists of a register file (64 32-bit single precision registers, that can be paired for double precision values), a program counter, a fixed-point ALU, and an instruction sequencer. Thread units are simple processors that issue and execute instructions in program order. Several threads share one data cache and one floating-point unit, forming a *thread group* (TG in Figure 1). The floating-point units are pipelined and can complete a multiply and an add in every cycle.

The Cyclops architecture defines a 3-operand, load-store RISC-like ISA with approximately 60 instruction types. For designing the Cyclops ISA we selected the most widely used instructions of the Power-PC architecture, to which we added multithreaded functionality, such as atomic read-modify-write memory operations. Most instructions execute in one cycle. Each thread can issue an instruction in every cycle, if resources are available and there are no dependences with previous instructions. If two threads try to issue instructions using the same shared resource, one thread is selected as winner in a round-robin scheme to prevent starvation. If an instruction cannot be issued, the thread unit stalls until all resources become available, either through the completion of previously issued instructions, or through the release of resources held by other threads.

The architecture itself does not specify the exact number of components in a chip and supports a multitude of design points. As a starting point of our studies we adopt the configuration listed in Table 1. Table 1(a) shows the instruction latencies for this configuration. The delay for an instruction is decomposed into two parts: *execution* (E) cycles and *latency* (L) cycles. Each functional unit can begin processing a new instruction after E cycles. The result is available after $E + L$ cycles, and, since the functional units are pipelined, they can be utilized by other instructions during the latency period. For memory operations, the latency of the operations depend how deep into the memory hierarchy we have to go to fetch the result.

The relative number of components in Table 1(b) is determined by silicon area constraints and most common instruction type percentages. We expect these numbers to change as manufacturing technology improves. The balance between different resources might also change as a consequence of particular target applications and as our understanding of the different trade-offs improves. The base architecture studied in this paper specifies 32 thread groups with their respective floating point units shared by one to eight threads. With a 500 MHz clock cycle, it translates into 1 GFlops peak performance per floating-point unit, or 32 GFlops peak performance per chip.

Each of the 32 data caches of 16 KB (one per thread group) has 64-byte lines and is 8-way set-associative. The data caches are shared among all threads in the chip. The architecture allows the software to implement an entire spectrum of cache configurations [4]. In this paper we consider only the simplest organization, in which the 32 caches behave as a single, multiported 512 KB cache with uniform latency.

Instruction caches are 32 KB, 8-way set-associative with 64-byte line size. In the base architecture, one instruction cache is shared by 2 thread groups. Unlike the data caches, the instruction caches are private to the threads in the thread groups. In addition, to improve instruction fetching, each thread contains a Prefetch Instruction Buffer (PIB).

Table 1. Design parameters for the reference Cyclops architecture.

(a) instructions

Instruction type	Execution	Latency
Branches	2	0
Integer multiplication	1	5
Integer divide	33	0
Floating point add, mult. and conv.	1	5
Floating point divide (double prec.)	1	30
Floating point square root (double prec.)	1	56
Floating point multiply-and-add	1	9
Cache hit	1	6
Cache miss	1	24
All other operations (except memory ops.)	1	0

(b) components

Component	# of units	Params/unit
Threads	1, . . . , 256	single issue, in-order, 500 MHz
FPUs	32	1 add, 1 multiply
D-cache	32	16 KB, 8-way assoc., 64-byte lines
I-cache	16	32 KB, 8-way assoc., 64-byte lines
Memory	16	512 KB

A large part of the silicon area in the Cyclops chip design is dedicated to memory. The reference design considered in this paper has 16 banks of on-chip memory shared between thread units. Each bank has 512 KB for a total of 8 MB of embedded memory. The banks provide a contiguous address space to the threads. Addresses are interleaved to provide higher memory bandwidth. The banks have uniform latency. The unit of access is a 32-byte block, and threads accessing two consecutive blocks in the same bank will see a lower latency (3 cycles vs. 9 cycles for the first block) in burst transfer mode. In the default configuration, the aggregated memory bandwidth is 40 GB/s.

Finally, there are some features of the Cyclops architecture that are not analyzed in this paper. These include off-chip memory, interchip communication [2], and intrachip synchronization hardware [4].

3. Software Environment

The Cyclops system software stack consists of a compiler, kernel and runtime libraries. Our environment exports a familiar (POSIX) interface without the complexity of a full Unix system residing on a node. Currently applications are parallelized by hand, using the pthread model to take advantage of the large number of threads in a chip.

The kernel, libraries, and applications are generated with a cross-compiler based on the GNU toolkit (version 2.95.3), re-targeted for the Cyclops instruction set architecture. This cross-compiler supports C, C++, and FORTRAN77.

The resident kernel supports single user, single process within a Cyclops chip. The kernel is small when compared with current operating systems and it is well adapted to the limited resources available in the chip, providing a thin interface to the hardware for communication, synchronization, timers, and interrupts. Its simplicity also delivers high performance since it does not require traversing many layers of software to access the hardware. The kernel exposes a single-address space shared by all threads. Due to the small address space and large number of hardware threads available, no resource virtualization is performed in software: virtual addresses map directly to physical addresses (no paging) and software threads map directly to hardware threads. Every software thread is preallocated with a fixed size stack per thread (selected at boot time), resulting in fast thread creation and reuse.

We have developed an architecturally accurate simulator that interprets kernel and application code generated by the Cyclops compiler and models the microarchitecture of the Cyclops processors. Although our simulator is not cycle-accurate, it estimates performance by modeling latencies and contention for resources at all levels of the Cyclops hierarchy. The simulator is parametrized such that different architectural features can be specified at program execution. The parameters shown in Table 1 are varied to obtain the performance results presented in Section 5. The simulator produces instruction traces, instruction histograms, and resource utilization statistics, such as floating point unit usage and contention, cache hit and miss ratio, memory accesses and contention.

4. Description of the Benchmark Suite

We can leverage the unconventional characteristics of the Cyclops architecture (limited memory, unbalanced processing to memory ratio, large number of registers and threads, and shared functional units) by carefully tuning applications. In the past we have experimented with standard benchmarks such as STREAMS and Splash-2 [4] and ported complete applications that run on large Cyclops systems [2]. The complexity of the interactions between the code, compiler, runtime libraries, and architecture prevented us from quantifying the effect on performance of each design feature.

In this paper we evaluate the performance of Cyclops using a custom built toolkit. We present results for five scientific kernels: FFT, Finite Differences, matrix multiplication, sparse matrix-vector product and Cholesky factorization. The simplicity of these codes allows for a better understanding of their performance characteristics, which increases our insight into the behavior of the architecture.

The results in Section 5 describe several implementations of each benchmark. The *naive* version of a benchmark is a simple implementation compiled with the `gcc` compiler using the basic optimization option `-O2`. *Naive* *unroll* uses the same naive source codes but is compiled to take advantage of the loop unrolling features of gcc (`-O2 -funroll-all-loops`). In addition, we provided manually optimized versions of our benchmarks using well known optimizations such as unrolling and register tiling. We did not perform cache blocking optimizations on these codes.

We designed the data set size for each benchmark in such a way as to exceed the cache capacity. For two of the benchmarks we also provided a small data set size that does fit in the cache.

FFT is a modified version of the Splash-2 FFT benchmark. This multi-threaded implementation maps a vector of n complex numbers into a two-dimensional array. Every thread computes the FFT on an equal number of rows of this matrix for $\log n/2$ phases. The matrix is then transposed in parallel and the threads complete the FFT on the remaining $\log n/2$ phases.

The difference between our implementation and the one provided by Splash-2 is that we perform a simple transpose procedure rather than a more involved blocked transpose. The computation of the FFTs in each phase is completely local to each thread. The only synchronization primitive used in this benchmark is a global barrier in which threads wait before transposing the matrix.

Manually optimized versions: we unrolled the inner loop of the FFT computation by factors of 2, 3 and 4, and also unrolled the inner loop of the transpose procedure four times.

Data set size: the FFT benchmark was executed with a single data set, a vector of 64K complex numbers, and requires 4.01 MB of storage.

MM (matrix multiplication) computes $AB = C$ with $A_{m \times p}$, $B_{p \times n}$ and $C_{m \times n}$. Our multithreaded implementation partitions the matrix C into $r \times s$ block matrices using $t = r \times s$ threads and assigns each submatrix to a single thread. Thus each thread computes at most $\lceil \frac{m}{r} \rceil \times \lceil \frac{n}{s} \rceil$ dot products of size p.

Manually optimized versions: loops were register tiled, reordered and unrolled in order to achieve maximum register reuse. We ran several *manualXY* implementations, where X and Y stand for the degree of unrolling of the two outer loops.

Data set size: we ran this benchmark using two problem sizes: a *small* size $m \times n \times p = 192 \times 192 \times 100$, requiring 0.57 MB of storage, which fits into cache, and a *large* $384 \times 384 \times 200$ problem that requires 2.29 MB of memory, and does not fit into cache.

SPARSE is the multiplication of a sparse matrix S by a vector, is the main kernel of many iterative linear solvers. Our implementation represents the sparse matrix S using *row-indexed sparse storage* [13, 15]. A fill parameter f controls the sparsity of the matrix.

The inner loop of the sparse-matrix vector product requires three memory loads, two of which are in an indirect subscript expression $S(i(j))$, for every non-zero element of the sparse matrix S. The location of the load for the indirect access is particularly difficult to predict and therefore the latency is difficult to hide. Consequently sparse-matrix vector codes generally suffer from poor performance.

In our parallel implementation the rows of the matrix S and the solution vector y are partitioned between threads. This implementation does not require thread synchronization.

Manually optimized versions: each thread multiplies one or two rows of the matrix at the same time, and in the manual implementation the inner loop is unrolled 8 times, hence the names *manual18* and *manual28*.

Data set size: this benchmark was run with matrix size 1024×1024 and fill factor $f = 4$, requiring 3.03 MB of main memory.

FD (finite-differences) computes the solution to the two-dimensional Poisson equation $\nabla^2 u = f$, where $f(x, y)$ is a random function, in the square domain $\Omega = (0, 1)^2$ with Dirichlet boundary conditions $u = 0$. FD uses a standard five-point stencil on an $m \times n$ regular mesh. The solution to the linear approximation by Jacobi iteration uses two alternating matrices $\hat{u}^k$ and $\hat{u}^{k-1}$. Like in the matrix multiplication kernel, we partition the threads into an array of r by s threads. At every iteration k, every thread is responsible to compute at most $\lceil \frac{m}{r} \rceil \times \lceil \frac{n}{s} \rceil$ unknowns in the matrix $\hat{u}^k$ using the previously computed values in $\hat{u}^{k-1}$. Threads synchronize in a global barrier before proceeding to the next iteration. For simplicity, the benchmark runs for a fixed number of iterations rather than testing for convergence.

Manually optimized versions: in the *manualXY* implementation threads compute the stencil in X by Y blocks, resulting in better register locality. X and Y are limited by the size of the register file.

Data set size: this benchmark was run with both small ($m = n = 128$) and large ($m = n = 512$) data sizes, requiring 0.38 MBytes and 6.04 MBytes of storage, respectively.

Cholesky is a thread-parallel version of the blocked Cholesky factorization. The triangular input matrix is subdivided into blocks of size $n = 4$ or $n = 8$. In each iteration the algorithm solves a single block and updates the rest of the matrix. The single block is always solved by a single thread, whereas the update is distributed among as many threads as possible. Each iteration requires three global barriers.

Table 2. Benchmark performance: MFlops on a 750 MFlops IBM PowerPC 604e workstation. The best performance of each benchmark is highlighted.

benchmark	size	description	storage	gcc		xlc		Best
			(MB)	naive	manual	naive	manual	% of peak
FFT	large	1D FFT of 64K complex	4.01	78.15	83.01	81.21	**84.90**	11.32
MM	small	192x192x100	0.57	25.13	**144.56**	25.42	137.81	19.27
	large	384x384x200	2.29	25.21	120.02	22.17	**125.11**	16.68
	L1 PPC	32x36x35	0.28	216.00	537.60	265.85	**556.14**	74.15
SPARSE	large	1024 x 1024	3.03	29.45	**31.02**	28.65	29.62	4.14
FD	small	128 x 128	0.38	**43.12**	40.76	37.58	38.28	5.75
	large	512 x 512	6.04	**23.00**	21.45	**23.00**	21.35	3.06
	L1 PPC	33 x 33	0.26	125.65	**226.88**	124.69	209.42	30.25
Cholesky	large	880 x 880	6.19	27.98	80.81	32.13	83.17	11.08
	L1 PPC	240 x 240	0.46	46.08	153.41	57.60	**230.11**	30.68

Manually optimized versions: our implementation of the benchmark completely unrolls operations on the individual blocks, and register tiles the update operation (which is in fact a series of matrix multiplications). Whereas the *naive* version has block size 4, there are two manually optimized versions with block sizes of 4 and 8.

Data set size: this benchmark was run with a matrix of size 880 × 880, requiring 6.19 MBytes of storage.

Benchmark Performance on PowerPC: Register tiling and unrolling optimizations are necessary to obtain good performance in Cyclops, but they apply equally well to more conventional RISC architectures. For comparison, we ran the benchmarks on a 375MHz PowerPC 604e machine with a peak performance of 750 MFlops. These results are shown in Table 2. We used both `xlc`, the native compiler, and `gcc`. We also added a data set size that fits into the PowerPC L1 cache. These implementations are not optimized for PowerPC. The last column of Table 2 shows the percentage of peak performance obtained by the benchmarks in the best compiler/implementation combination.

5. Experimental results

The experimental results presented in this section show the performance effect of adjusting several parameters of the Cyclops architecture: the number of threads per FPU, cache latency and memory bandwidth. We performed these experiments to gain a better understanding of the design trade-offs for this architecture.

Increasing the degree of parallelism in the architecture, by having multiple threads sharing functional units, results in better performance because it increases the utilization of the functional units. More thread units mean a larger total number of registers and data fetching units, and hence better tolerance of cache and memory latency. We measured the latency that these benchmarks can tolerate by varying the cache latency in our simulator. We show that there is no major impact on performance until latency reaches at least 30 cycles. This gives a reasonable target to the designers of the memory subsystem and allows for resources to be better spent elsewhere. More simultaneous memory requests result in higher pressure on the memory system. We show that our design point of 40 GBytes/s is adequate for most applications.

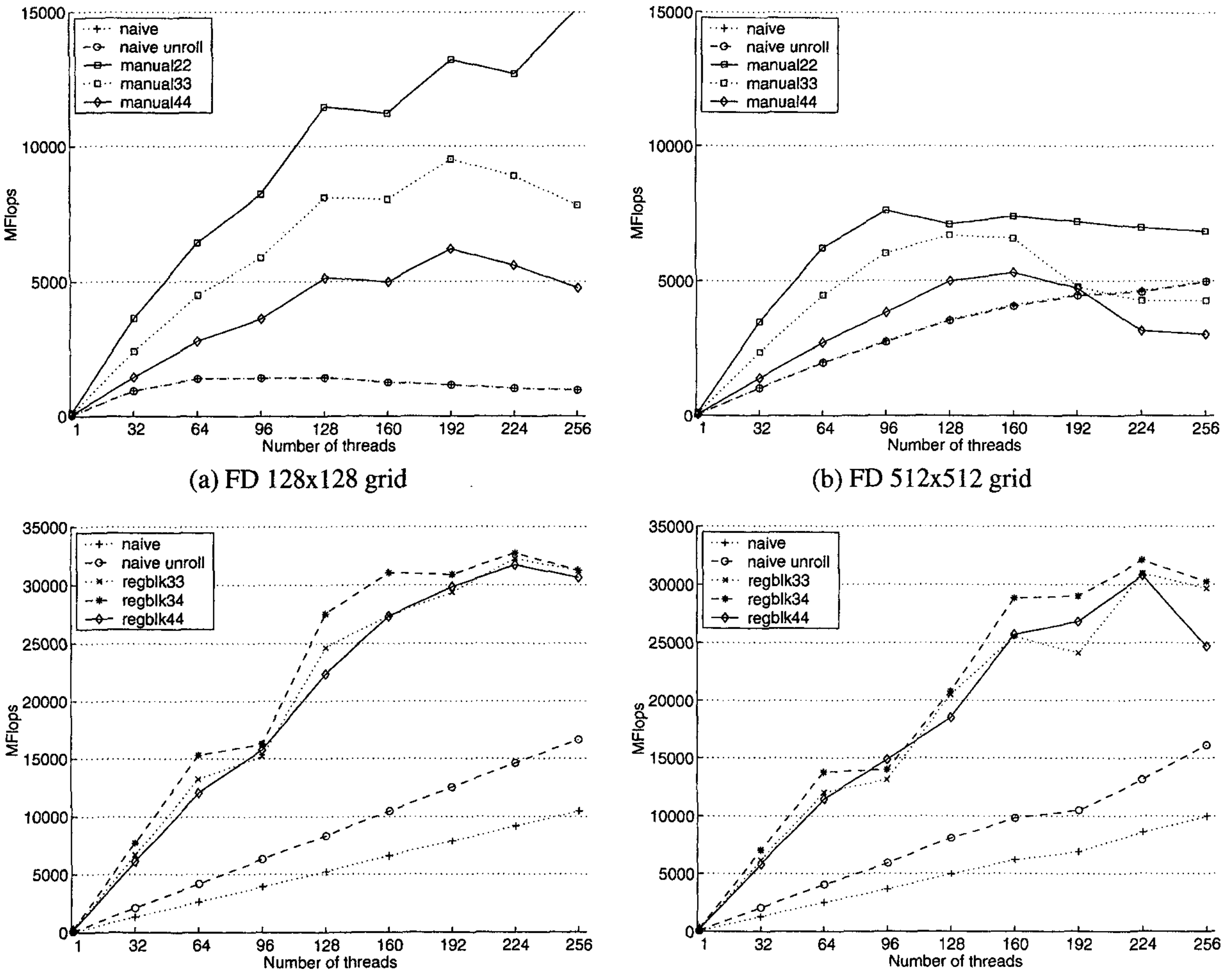
All experiments are performed on a chip with constant peak performance (32 GFlops) given by 32 FPUs running at 500 MHz. We vary the number of threads between 1 and 8 per FPU (for a total number of threads between 32 and 256 in a chip). As a reference, we also present the performance of a single-threaded Cyclops chip (with a single FPU).

5.1. Baseline Results

The tests shown in Figures 2 and 3 quantify the effect on performance of compiler and hand optimizations, in particular loop unrolling and register tiling. These tests are run on the base Cyclops configuration described in Section 2, with a variable number of threads serving an FPU.

The graphs in Figures 2 and 3 share a common pattern: linear scaling up to the maximum performance, followed by a plateau or a degradation of performance. The performance plateau for each benchmark is determined by architectural features that the benchmark saturates.

- **Peak floating point performance:** a benchmark can achieve peak floating point performance only if both the adder and the multiplier in the FPU are utilized 100%, which may not be achievable when e.g. the



(a) FD 128x128 grid

(b) FD 512x512 grid

(c) MM 192x192x100

(d) MM 384x384x200

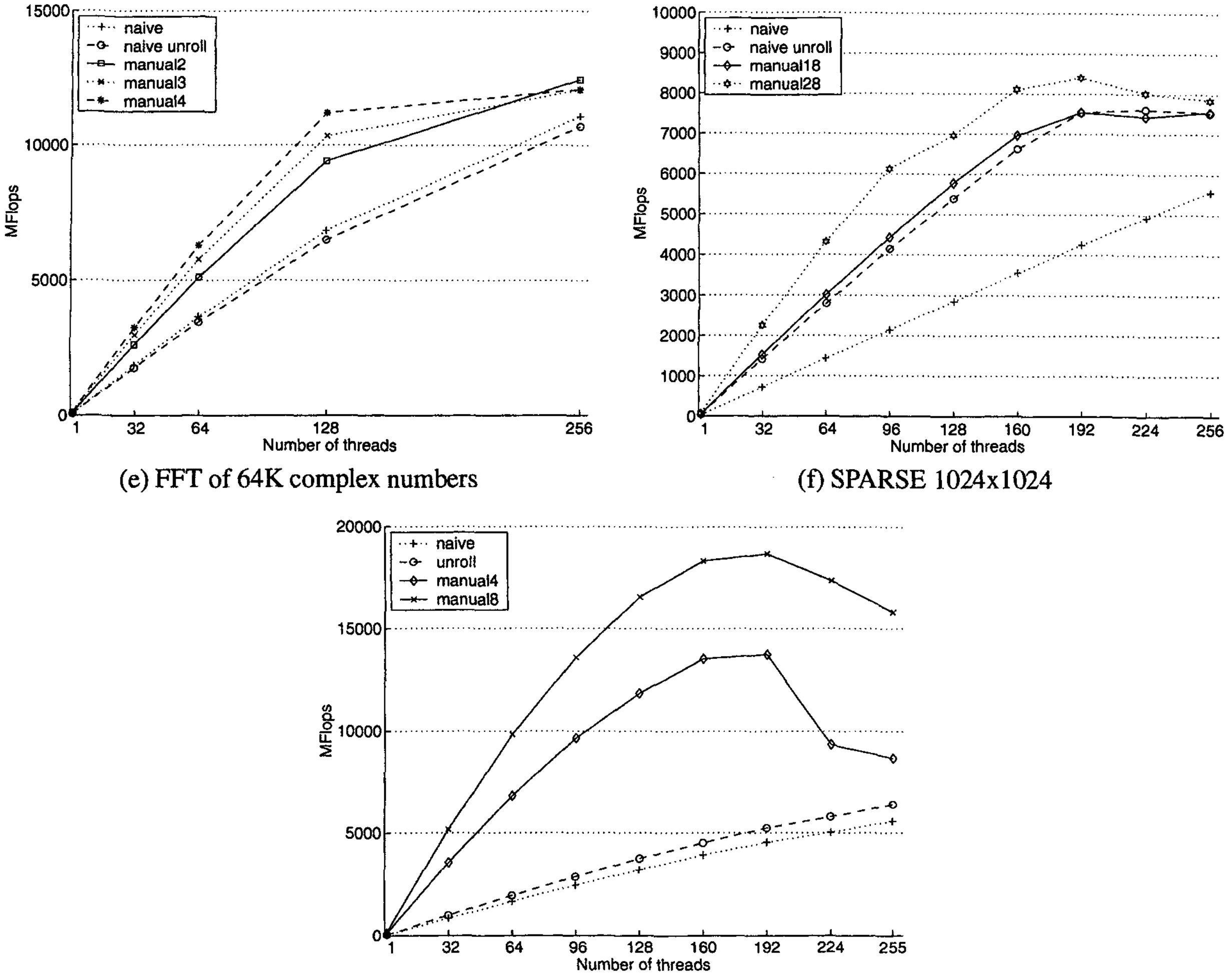
Figure 2. Performance obtained by naive, naive unroll, and manual unroll on the default Cyclops configuration

calculation involves a larger number of additions than multiplications. A benchmark's theoretical peak floating point performance is typically a constant fraction of the theoretical peak of the machine.

- **Memory bandwidth:** the theoretical peak performance of a benchmark is also limited by the ratio of memory to FP operations in a computational kernel's innermost loop. The default configuration of the Cyclops architecture provides 40 GBytes/s of memory bandwidth. Given the ratio of memory accesses for every FP operation, r, we can estimate the peak memory bandwidth performance as $P_{\text{MEM}} = \frac{40}{r}$. For the purpose of this formula we only need to consider memory accesses that are cache misses.

- **Cache latency:** the time it takes to load each memory reference into a register affects the time when an FP operation is scheduled. More latency introduces bubbles into the FPU's pipeline and reduces its utilization. Latency can be countered by increasing the number of threads feeding the FPU which increases its degree of utilization. Because of this the effect of cache latency tends to show up as linear increase in performance with increasing number of threads per FPU, which flattens out when the effect of cache latency has been mitigated.

Next we discuss how these parameters affect each of our kernels' performance.



(e) FFT of 64K complex numbers

(f) SPARSE 1024x1024

(g) Cholesky 880x880

Figure 3. Performance obtained by naive, naive unroll and manual unroll on the default Cyclops configuration

The FD benchmark Performance results for this benchmark are shown by Figure 2 (a) and (b), corresponding to the small and large data sets, respectively. Each figure contains plots for five different levels of code optimization: *naive* and *naive unrolled*, corresponding to compiler optimizations explained in Section 4, and *manual22*, *manual33* and *manual44* versions representing manual optimizations obtained by tiling the two nested loops of the FD kernel and partially overlapping an $n \times n$ section of five-point stencils in a single iteration, where $n = \{2, 3, 4\}$. The *manual22* version of the code shows better performance than *manual33* and *manual44*, because the latter ones cause the compiler to spill registers.

Figures 2 (a) and (b) show very different behavior. In Figure (a) the data set fits in the cache, and performance is limited by cache latency. Figure (b) shows the large data set, where the limiting factor is memory bandwidth. The 2×2 stencil of the *manual22* version of the code requires the loading/storing of 14 double precision numbers for every stencil, resulting in about 5.6 bytes of memory for each floating point operation. Thus the 40 GBytes/s peak memory bandwidth manifests itself as a plateau at $P_{\text{MEM}} = \frac{40}{5.6} = 7.14\,\text{GFlops}$, which is plainly visible on the figure.

The FD benchmark performs four times more floating point additions than multiplications, and cannot fill all available time slots for the multiplier. This is corroborated by the statistics reported by our simulator: in the single thread configuration the adder's utilization rate is 46.71% versus the multiplier's 20.32%.

The MM benchmark The results for MM, using the small and large data sets, are shown in Figure 2 (c) and (d) respectively. All manually register tiled versions of the code, *manual22*, *manual33*, *manual34* and *manual44* were better than the compiler optimized versions; of these, *manual34* performed best, because *manual44* caused the compiler to spill registers.

The similarity between the two MM figures is remarkable considering the difference between data set sizes. This is a result of MM being a computational-bound algorithm where memory bandwidth is not a factor: the register-tiled version of MM requires approximately 1 byte of memory data for each FP operation it executes.

Both MM data sets, *small* and *large*, achieve the theoretical peak floating point performance of 32 GFlops.

The FFT benchmark The FFT kernel comes in *naive* and several manually unrolled versions. The performance of FFT on the Cyclops architecture, shown in Figure 3 (e), is limited by a combination of memory accesses and peak floating point performance. The bandwidth requirements of the butterfly pattern are about 3 bytes of memory for every FP operation, which imposes a performance plateau of $P_{\text{MEM}} = \frac{40}{3}$, or about 16 GFlops.

The SPARSE benchmark The SPARSE benchmark, shown in Figure 3 (f), is limited both by cache latency and memory bandwidth. Latency has a high effect on the code because of the two dependent memory loads in the code, but this effect can be mitigated by manually unrolling the code. Bandwidth affects the kernel because of the sheer amount of memory consumed: about 4.5 bytes in cache misses for every FP operation in the *manual28* benchmark. This causes performance to be limited to $P_{\text{MEM}} = \frac{40}{4.5} = 8.8$ GFlops.

The Cholesky benchmark The Cholesky benchmark, shown in Figure 3 (g), is typically a computation-bound algorithm, but on Cyclops this kernel was affected by cache latency and an inability to efficiently make use of the available parallelism.

The performance of *naive* parallel Cholesky is poor and shows the effect of cache latency (linear speedup). The manually blocked and tiled versions of Cholesky, *manual4* and *manual8*, represent two different points of compromise between manual register tiling and parallelism. The insufficient amount of parallelism in the code causes a decline in performance as the number of threads increases. The Cholesky benchmark is also affected by the relatively poor performance of the `SQRT` and `FDIV` operations on Cyclops, due to their simple implementation. Even with these limitations the Cholesky benchmark achieves 18 GFlops.

5.2. Data cache latency

From the results in Section 5.1 we conclude that intelligent register usage and instruction scheduling are essential to obtain performance on the Cyclops architecture. These experiments also show that the benchmarks need many threads per FPU in order to hide cache or memory latency, as shown by the linear increase in performance obtained when adding thread units.

To further evaluate the sensitivity of our benchmarks to data cache latency, we selected the best-performing version of each benchmark from the baseline runs and re-measured its performance with various latencies. The plots in Figures 4 and 5 highlight the "knee" of the performance curve: the point where performance starts declining due to cache latency effects.

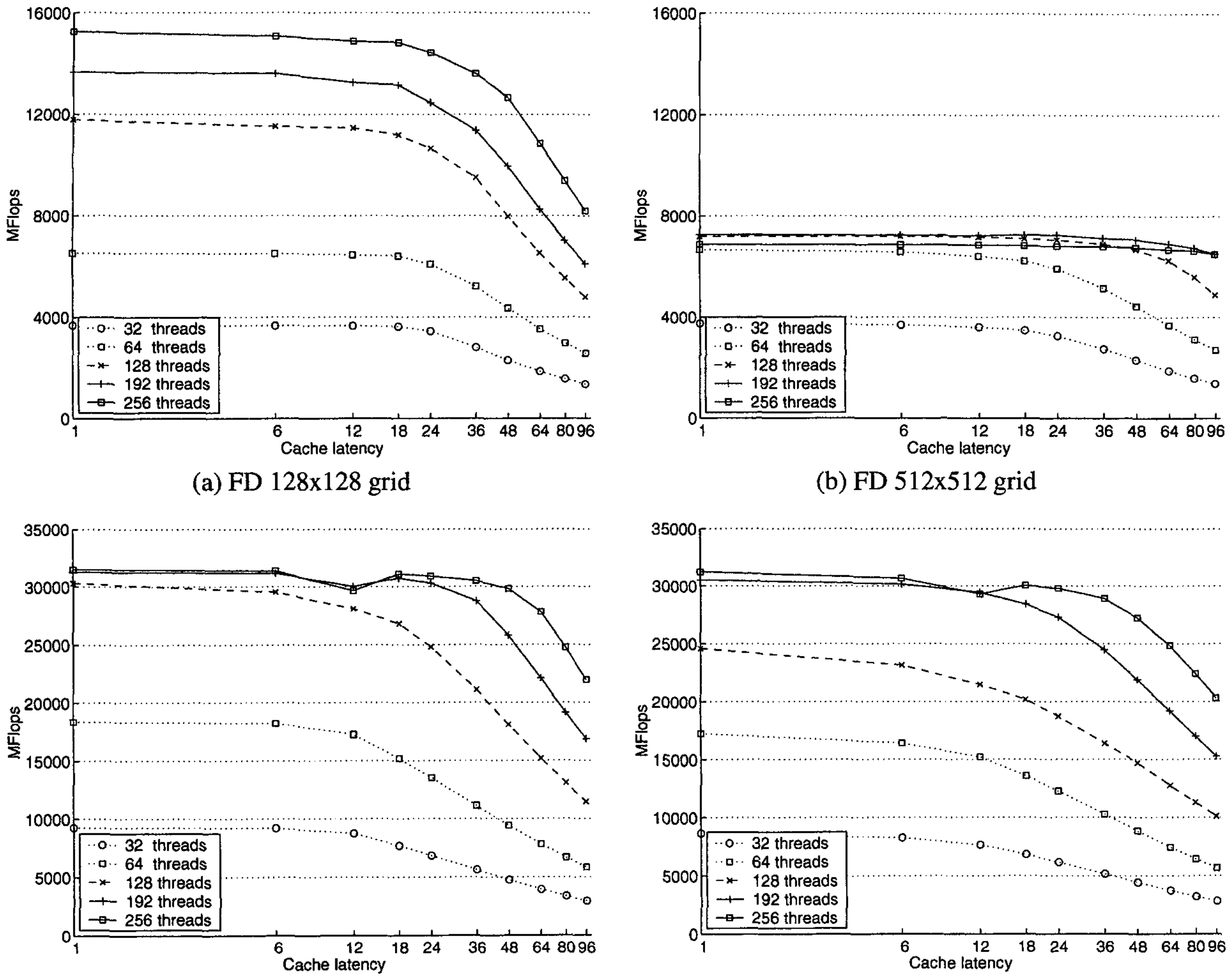
Using the location and characteristics of this knee we can categorize our benchmarks by their sensitivity to cache latency. Some of the benchmarks show remarkable tolerance: both versions of MM, large and small, shown in Figure 4 (c) and (d), tolerate large amounts of cache latency – up to 64 cycles without appreciable degradation in performance, given enough threads per FPU.

Even small cache latencies are not sufficient to achieve peak performance when only a few threads share an FPU. According to simulation data, threads in MM with the small data set on a one thread per FPU configuration stall 32% of the total number of cycles due to register file port conflicts and not because of register dependences. Our simulations use a two-ported register file, thus instructions like FMAD that access more than two registers have an extra execution cycle. At the same time, only 45% of the instructions are floating point operations. Because of these restrictions the FPUs are significantly under-utilized. The performance achieved in this case is limited to $32 \times 0.68 \times 0.45 \approx 9.2$ GFlops.

Benchmarks dominated by memory loads, like FFT and FD with the large data set, are less affected by cache latency variations since most memory accesses are cache misses. The performance knee tends to be less pronounced for these applications.

The three benchmarks most affected by cache latency are the FD benchmark running on the small data set, the SPARSE benchmark, and Cholesky. FD, shown in Figure 4 (a), is affected because of the relatively low degree of unrolling (2 × 2) applied to it; SPARSE, shown in Figure 5 (f), has a chain of two dependent memory loads in the innermost loop causing a delay of two cache latencies in every iteration; and Cholesky, shown in Figure 5 (g), is also sensitive to cache latency.

The cache latency measurements show that most of our benchmarks exhibit high tolerance to latency. With this design, having no cache at all is not unthinkable: at the very



(a) FD 128x128 grid

(b) FD 512x512 grid

(c) MM 192x192x100

(d) MM 384x384x200

Figure 4. Performance vs. cache latency, manually unrolled benchmarks

least, for the applications studied, Cyclops could place the caches on the memory side and dispense with the inherent complexity of cache coherence.

5.3. Memory bandwidth

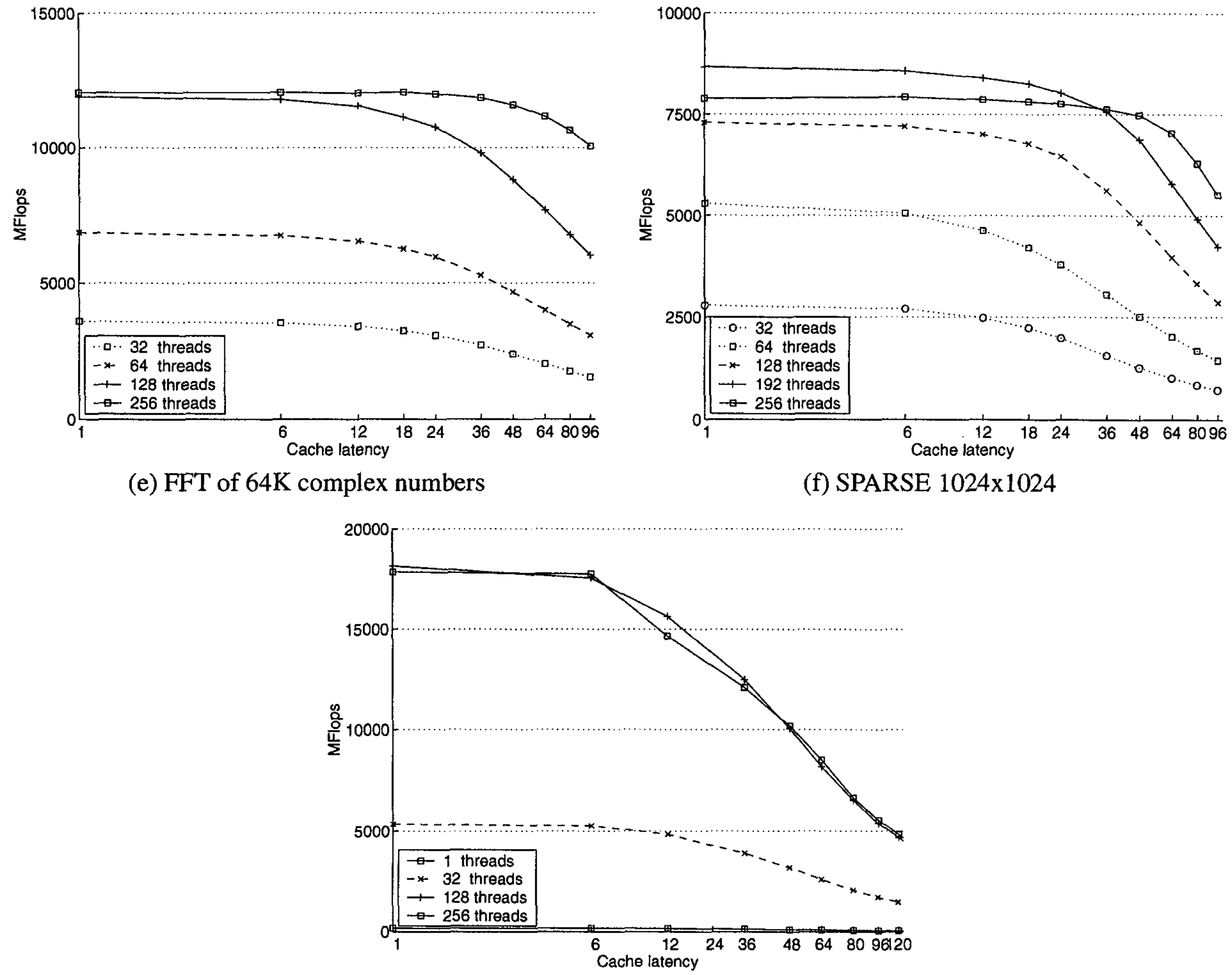
Figures 6 and 7 show the effect of memory bandwidth on the performance of our best manual implementation of each kernel. Three curves in each figure plot Cyclops configurations with 16, 32 and 64 banks of 512 KB, 256 KB and 128 KB respectively maintaining a constant 8 MB of embedded DRAM in a chip. With these configurations the total bandwidth of the memory system is 40 GBytes/s, 80 GBytes/s and 160 GBytes/s respectively.

Figures 6 and 7 can be used to identify the benchmarks with high memory bandwidth requirements: their performance improves when bandwidth is increased. These benchmarks are FD with the large data set, FFT and SPARSE. All other benchmarks show little or no variation when bandwidth changes, because they are dominated by floating point operations (like MM), by cache latency (FD, small data set) or by data parallelism (Cholesky).

As discussed earlier, the FD benchmark requires about 5.6 bytes of memory for each FP operation. For bandwidth limits of 40, 80 and 160 GBytes/s, this results in P_{MEM} values of 7.1, 14.2 and 28.4 GFlops respectively, the first two of which can be observed clearly on Figure 6 (b); the third cannot be observed because at this memory bandwidth the benchmark hits the cache latency limit first.

The FFT benchmark requires only about 3 bytes of memory per FP operation, resulting in P_{MEM} values of 16, 32 and 64 GFlops respectively; only the first of these can be seen in Figure 7 (e) for the same reason as FD.

The SPARSE benchmark requires 4.5 bytes/FP opera-



(e) FFT of 64K complex numbers

(f) SPARSE 1024x1024

(g) Cholesky 880x880

Figure 5. Performance vs. cache latency, manually unrolled benchmarks

tion, resulting in P_{MEM} values of 8.8, 18 and 36 GFlops respectively; only the first of these can be observed in Figure 7(f).

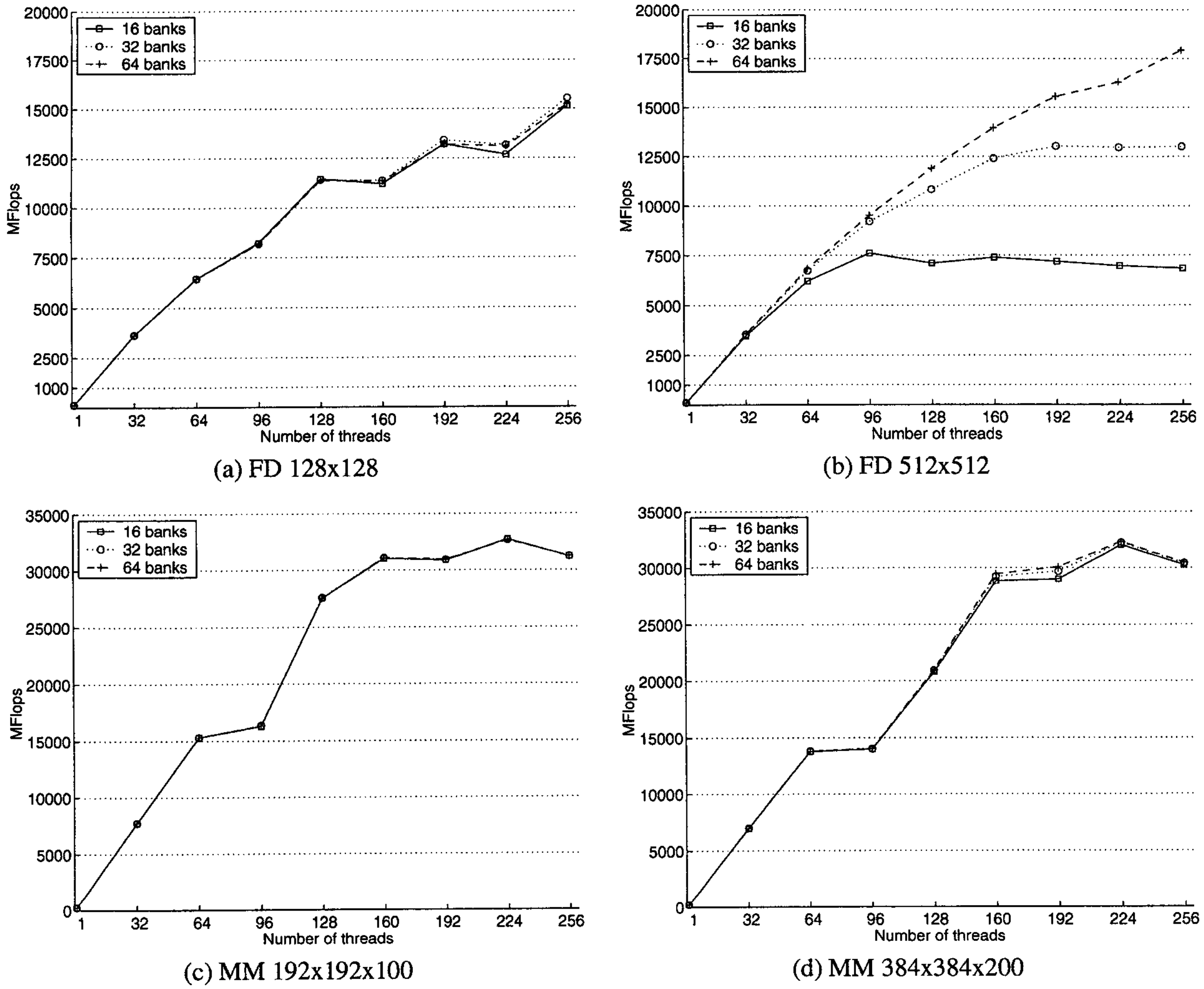
In conclusion, the balance in Cyclops between available memory bandwidth and floating point performance should be driven by the target application; for most of the benchmarks the 40 GBytes/s seems reasonable, but a few of the benchmarks could make use of higher bandwidth. Quadrupling the bandwidth to 160 GBytes/s does not seem necessary, at least for these applications.

6. Related Work

Our design for Cyclops is ambitious, but within the realm of current or near-future silicon technology. Combined logic-memory microelectronics processes will soon deliver chips with hundreds of millions of transistors. Several research groups have advanced processor-in-memory designs that rely on that technology. We discuss some of the projects that are related to Cyclops.

The MIT RAW architecture [1, 21] consists of a highly parallel VLSI design that fully exposes all hardware details to the compiler. The chip consists of a set of interconnected tiles, each tile implementing a block of memory, functional units, and switch for interconnect. The interconnect network has dynamic message routing and a programmable switch. The RAW architecture does not implement a fixed instruction set architecture (ISA). Instead, it relies on compiler technology to map applications to hardware in a manner that optimizes the allocation of resources.

Architectures that integrate processors and memories on the same chip are called Processor-In-Memory (PIM) or Intelligent Memory architectures. They have been spurred by technological advances that enable the integration of



(a) FD 128x128

(b) FD 512x512

(c) MM 192x192x100

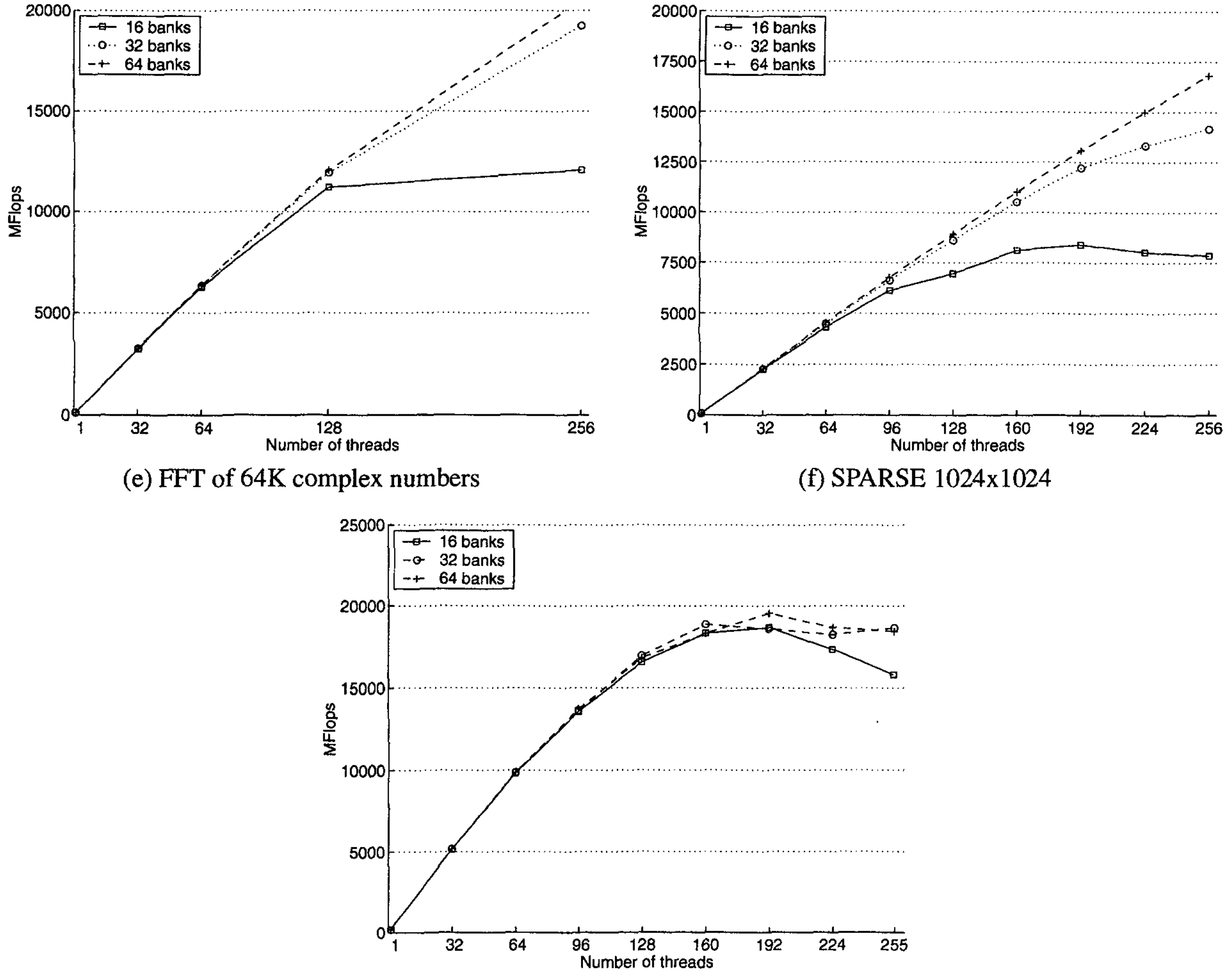
(d) MM 384x384x200

Figure 6. The effect of varying memory bandwidth on benchmark performance

compute logic and memory on a single chip. These architectures deliver higher performance by reducing the latency and increasing the bandwidth of processor-memory communication. Examples of such architectures are EXECUBE [9], IRAM [12], Shamrock [8], Imagine [14], FlexRAM [7, 18], DIVA [6], Active Pages [11], Gilgamesh [22] and MAJC [19]. The PIM chip is used as a coprocessor (Imagine, FlexRAM), or as the main engine in the machine (IRAM, MAJC, Piranha, Shamrock), or as a "cell" in a larger system (MIT RAW, EXECUBE and Cyclops). Another classification could be based on the number and type of the processors: FlexRAM and Imagine include many (more than 32) relatively simple processors, while EXECUBE, IRAM, MAJC, Piranha [3] and Shamrock include only a few (4-8). Cyclops goes beyond what has been proposed, using hundreds of processors.

Simultaneous multithreading exploits both instruction-level and thread-level parallelism by issuing instructions from different threads in the same cycle. It was shown to be a more effective approach to improve resource utilization than superscalar execution. Results presented in [5, 20] support our work by showing that there is not enough instruction-level parallelism in a single thread of execution, therefore it is more efficient to execute multiple threads concurrently.

The Tera MTA [16, 17] is another example of a modern architecture that tolerates latencies through massive parallelism. In the case of Tera, 128 thread contexts share the execution hardware. This contrasts with Cyclops, in which each thread has its own execution hardware. Both architectures can tolerate long latencies, and while Tera does not implement caches at all, we have shown that Cyclops can



(e) FFT of 64K complex numbers

(f) SPARSE 1024x1024

(g) Cholesky 880x880

Figure 7. The effect of varying memory bandwidth on benchmark performance

be made a cache-less machine.

7. Conclusions

We need to discuss two important limitations of the Cyclops architecture. First, combining logic and memory processes have a negative impact: the logic is not as fast as in a pure logic process and the memory is not as dense as in a pure memory process. For Cyclops to be successful we need to demonstrate that the benefits of this single-chip integration, such as improved memory bandwidth, outweigh the disadvantages. Second, due to its single-chip nature, Cyclops is a small-memory system. The off-chip memory is not directly addressable and its bandwidth is much lower. We can expect future generations of Cyclops to include larger memory. Nevertheless, the current ratio of 250 bytes of storage to 1 MFlops of compute power (compared to approximately 1 MB/1 MFlops in conventional machines) will tend to decrease.

The result is that Cyclops systems are not single purpose machines such as MD-Grape [10] but are not truly general purpose computers either. Our architecture targets problems that exhibit two important characteristics: massive amounts of parallelism and intensive computation. Examples of applications that match these requirements are molecular dynamics [2], raytracing, data mining, and linear algebra.

The corollary is that porting applications to Cyclops is not a simple process. The large number of threads and the limitations on caches and memory are not easily modeled by any compiler in existence, although some compiler optimizations (like cache tiling) address at least part of the problem. Our experiments show that in order to achieve good performance on Cyclops we need to exercise as many threads in the system as possible. This is partially a compiler problem, but it will definitely impact the models used to program this system. We foresee the use of parallel programming models such as OpenMP and MPI, as well as the simpler *pthreads* model we used.

We also show that, when the multithreaded architecture is used to its potential, it can hide large memory latencies, resulting in a system design that eschews cache coherence issues and may be able to do without caches at all.

Another issue that has surfaced in our experiments is that of memory bandwidth. When driving the FPUs to the limit, some of the applications stress memory more than we had predicted, resulting in performance degradation. This will need to be corrected.

Finally, we should emphasize that the results presented in this paper were obtained through simulation. Although we are confident of the general trends demonstrated, the results need to be validated through real measurements in hardware. Moreover, another design step is needed, setting up a compromise between the architectural features we want in the chip and the memory area limitations. As we proceed to complete the design of Cyclops and build prototypes, we will have the capability to perform the measurements and finalize the architecture.

References

[1] A. Agarwal. Raw computation. *Scientific American*, August 1999.

[2] G. S. Almasi, C. Caşcaval, J. G. Castaños, M. Denneau, W. Donath, M. Eleftheriou, M. Giampapa, H. Ho, D. Lieber, J. E. Moreira, D. Newns, M. Snir, and H. S. Warren, Jr. Demonstrating the scalability of a molecular dynamics application on a Petaflop computer. In *Proceedings of the 2001 International Conference on Supercomputing*, pages 393–406, June 2001.

[3] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.

[4] C. Caşcaval, J. G. Castaños, L. Ceze, M. Denneau, M. Gupta, D. Lieber, J. E. Moreira, K. Strauss, and H. S. Warren, Jr. Evaluation of a multithreaded architecture for cellular computing. In *Proceedings of the 8th International Symposium on High Peformance Computer Architecture (HPCA)*, Feb 2002.

[5] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, and D. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, pages 12–18, September/October 1997.

[6] M. W. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCross, J. Brockman, W. Athas, A. Srivasava, V. Freech, J. Shin, , and J. Park. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Proceedings of SC99*, November 1999.

[7] Y. Kang, M. Huang, S.-M. Yoo, Z. Ge, D. Keen, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an advanced intelligent memory system. In *International Conference on Computer Design (ICCD)*, October 1999.

[8] P. Kogge, S. Bass, J. Brockman, D. Chen, and E. Sha. Pursuing a petaflop: Point designs for 100 TF computers using PIM technologies. In *Frontiers of Massively Parallel Computation Symposium*, 1996.

[9] P. M. Kogge. The EXECUBE approach to massively parallel processing. In *Intl. Conf. on Parallel Processing*, August 1994.

[10] MD Grape project. http://www.research.ibm.com/grape.

[11] M. Oskin, F. T. Chong, and T. Sherwood. Active pages: A computation model for intelligent memory. In *International Symposium on Computer Architecture*, pages 192–203, 1998.

[12] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM: IRAM. In *Proceedings of IEEE Micro*, April 1997.

[13] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipies in C.* Cambridge University Press, 1992.

[14] S. Rixner, W. Dally, U. Kapasi, B. Khailany, A. Lopez-Lagunas, P. Mattson, and J. Owens. A bandwidth-efficient architecture for media processing. In *31st International Symposium on Microarchitecture*, November 1998.

[15] Scientific Computing Associates, Inc. *PCGPACK user's guide.*

[16] A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchel, J. Feo, and B. Koblenz. Multiprocessor performance on the Tera MTA. In *Proceedings Supercomputing '98*, Orlando, Florida, Nov. 7-13 1998.

[17] A. Snavely, G. Johnson, and J. Genetti. Data intensive volume visualization on the Tera MTA and Cray T3E. In *Proceedings of the High Performance Computing Symposium - HPC '99*, pages 59–64, 1999.

[18] J. Torrellas, L. Yang, and A.-T. Nguyen. Toward a cost-effective DSM organization that exploits processor-memory integration. In *Sixth International Symposium on High-Performance Computer Architecture*, January 2000.

[19] M. Tremblay. MAJC: Microprocessor architecture for Java computing. Presentation at Hot Chips, August 1999.

[20] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[21] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, pages 86–93, September 1997.

[22] H. P. Zima and T. Sterling. The Gilgamesh processor-in-memory architecture and its execution model. In *Workshop on Compilers for Parallel Computers*, Edinburgh, Scotland, UK, June 2001.