Characteristics of AMD Barcelona Floating Point Execution

Ben Bales and Richard Barrett

Computer Science and Mathematics Division

Oak Ridge National Laboratory

Cray User Group

Atlanta, GA

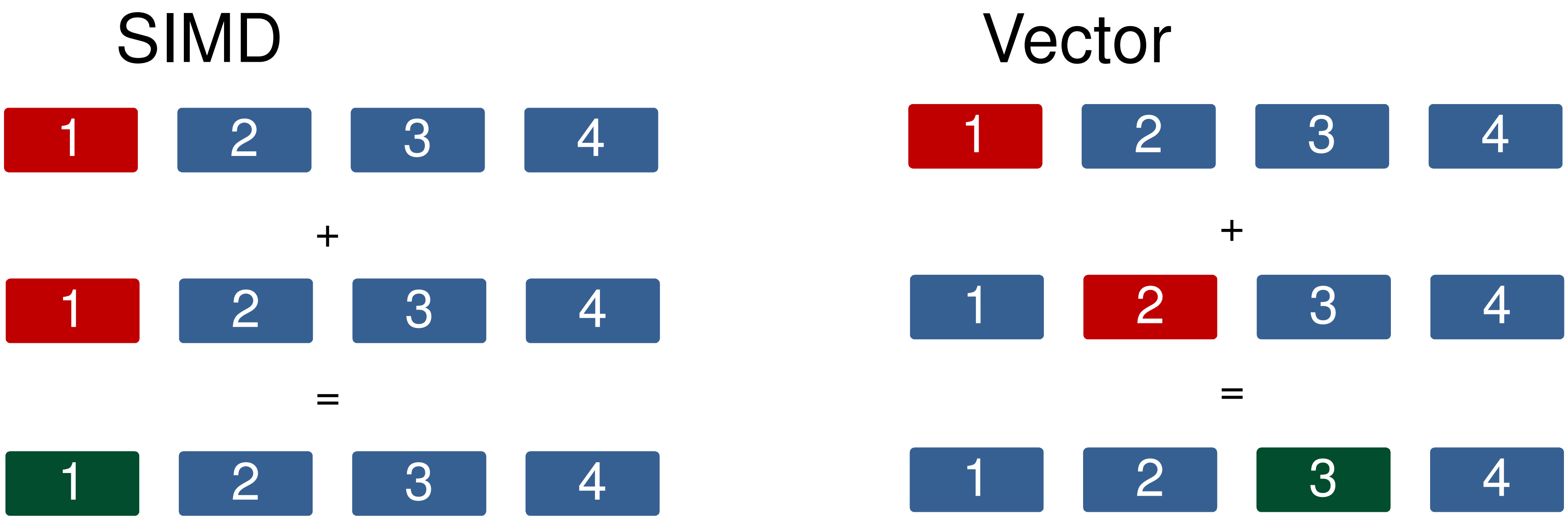
May 7, 2009

Managed by UT-Battelle for the
U. S. Department of Energy

Subjects

- Definitions
- Barcelona Overview
- Branch Predictor + Conditional Moves
- Software Pipelining
- Vector Operations
- How to Take Advantage of Given Optimizations

SIMD and Vector



Software Pipelining

In-Order Scheduler

Lookahead { Multiply, Multiply }

- Multiply
- Multiply
- Multiply
- Add
- Multiply
- Subtract
- Add
- Add
- Add

Out-of-Order Scheduler

Lookahead { Multiply, Multiply, Multiply, Add, Multiply, Subtract, Add }

- Multiply
- Multiply
- Multiply
- Add
- Multiply
- Subtract
- Add
- Add
- Add

<u>Software Pipelining references attempts to manually guarantee the existence of superscalar code within the Lookahead window.</u>

Barcelona Overview

Three level cache architecture

•L1/L2 – High speed computational caches (256 and 128bits per cycle respectively)

SIMD Instruction Set

•Limited "Vector" operations

•Three SIMD pipelines

<u>This presentation focuses on single precision</u>

Branches

Four sub-topics

- SSE conditional moves
- Integer comparators
- Branch prediction mechanism
- Case study: CORDIC Arctangent

SSE Conditional Moves

Branches that can be expressed as moves conditional on simple floating point comparisons SSE friendly

```
if(a < b) {
    sgn = 1.0;
} else {
    sgn = -1.0;
}

ans = c + sgn * d;
```

```
if(a < b) {
    ans = c + d;
} else {
    ans = c – d;
}
```

Integer Comparison Unit

As recommended in the AMD 10h optimization guide, floating point comparisons can be performed in integer units (chart taken from guide):

Comparison Against Zero	
Integer Comparison	**Replaces**
if(*((unsigned int *)&f) > 0x8000000U)	if(f < 0.0f)
if(*((int *)&f) <= 0)	if(f <= 0.0f)
if(*((int *)&f) > 0)	if(f > 0.0f)
if(*((unsigned int *)&f) <= 0x8000000U)	if(f >= 0.0f)

Branch Predictor

Dynamic branches predicted based on 2-bit Global History Bimodal Counters

Value:	**0**	**1**	**2**	**3**
Action:	No Branch	No Branch	Branch	Branch

Counters updated as branch conditionals evaluated

Branch Predictor cont.

Heuristic improved by addressing GHBC with Global Branch History

Global Branch History				GHBC
0	1	0	➡	3
1	0	1	➡	0

Unfortunately, loops waste half of the available bits!

Branch Predictor cont.

...	0	1	0	1	0	1	1	1	1	1	0	1

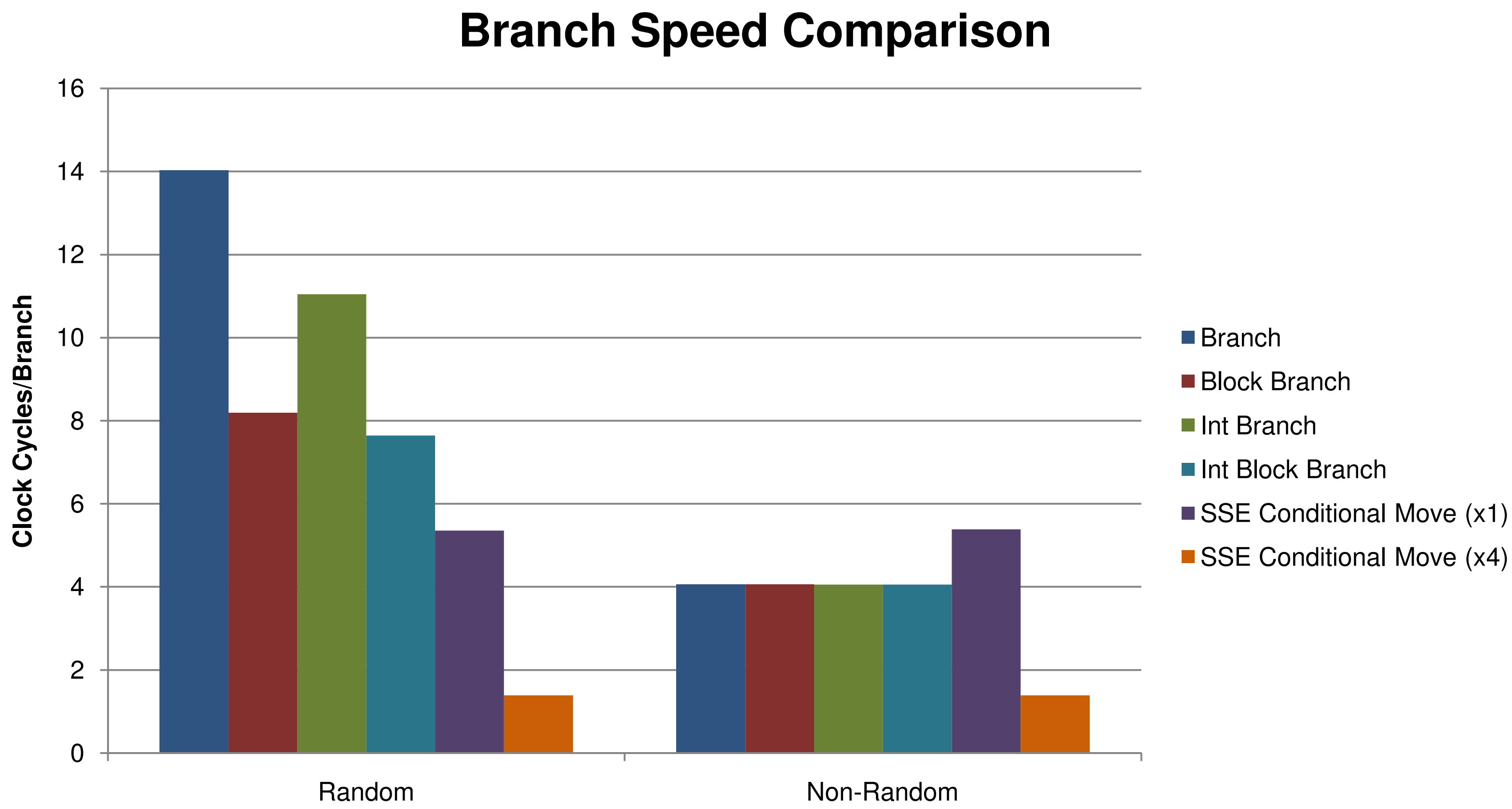
Dots – Instruction address

Red – Loop branch bits

Green – Legitimate branch bits

Unrolling loops can help improve the Legitimate/Loop ratio and make the GHBCs work better

Branch Benchmarks



CORDIC Arctangent

- CORDIC algorithms popular for implementing trig functions with limited hardware
- Basic algorithm is SSE friendly, but includes a tricky conditional move

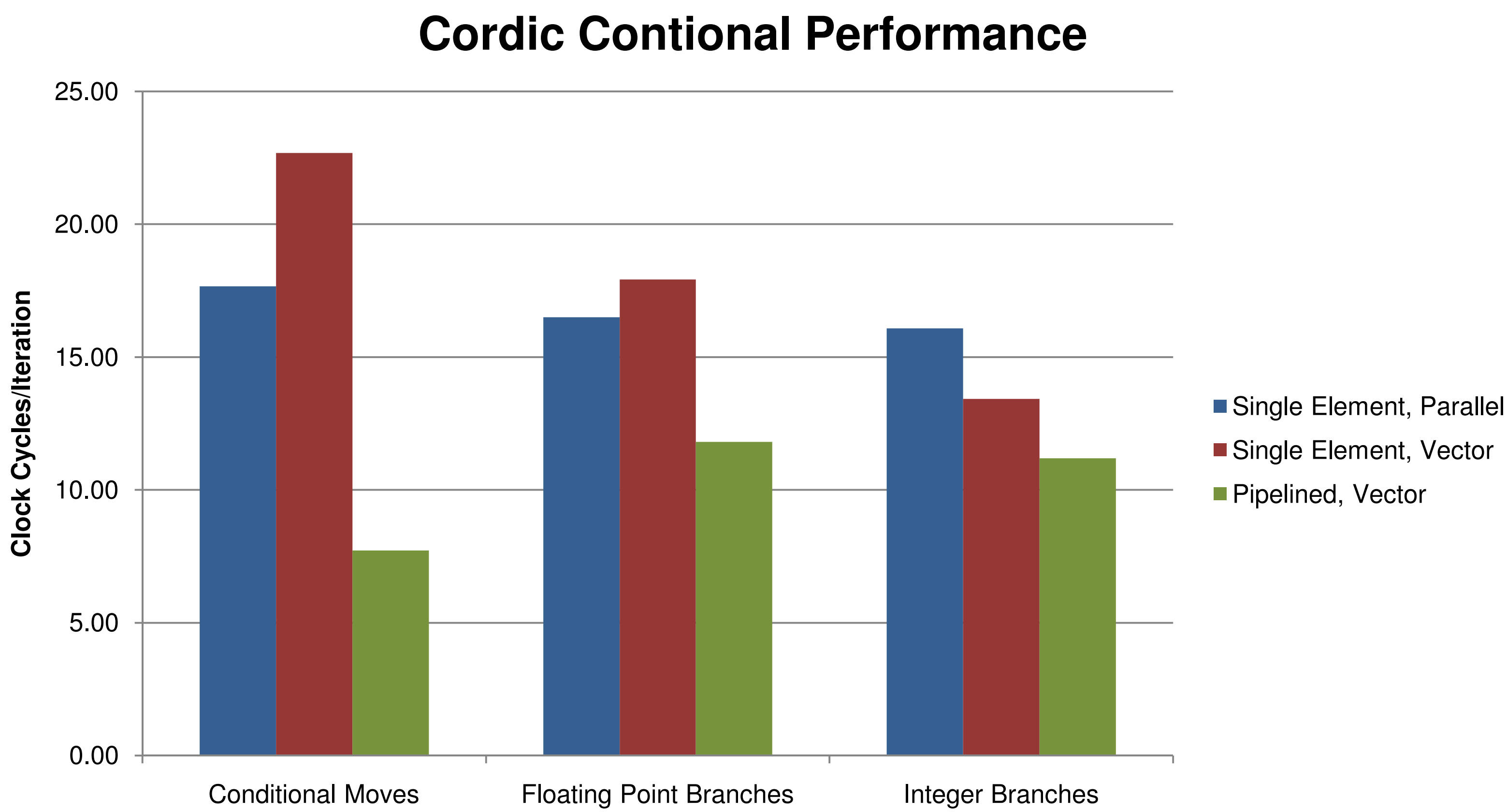
CORDIC Arctangent (SIMD)

```
for(i = 0; i < N; i++) {
        if(y < 0.0f) {
                s = 1.0f;
        } else {
                s = -1.0f;
        }

        n_x = x – s * y * powf(2.0f, -(float)i); //powf and atanf are
        n_y = y + s * x * powf(2.0f, -(float)i); //implemented in
        n_z = z – s * atanf(powf(2.0f, -(float)i); //lookup tables

        x = n_x;
        y = n_y;
        z = n_z;
}
```

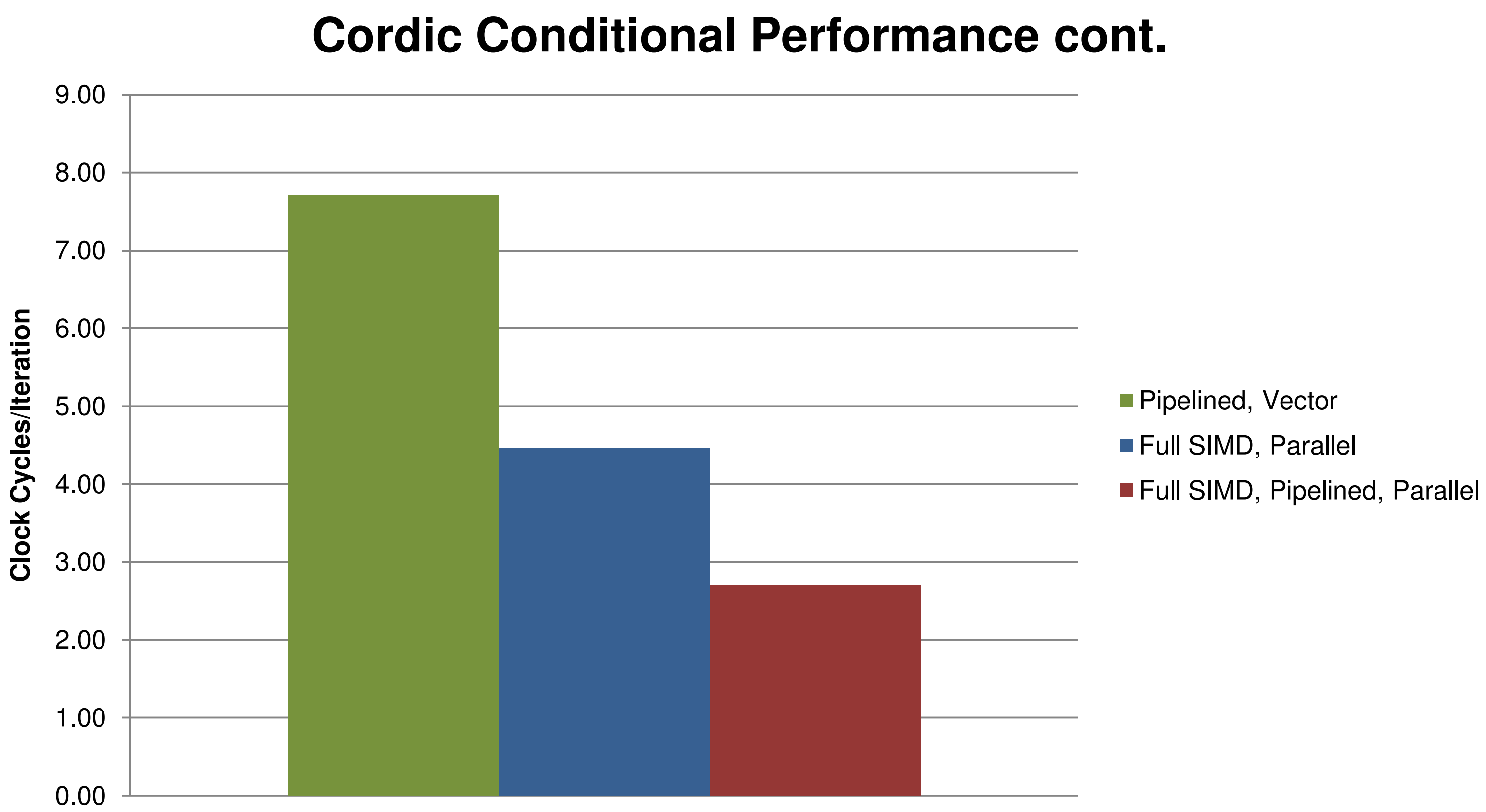
CORDIC Arctangent (Vector)

```
for(i = 0; i < N; i++) {
        if(y < 0.0f) {
                s = 1.0f;
        } else {
                s = -1.0f;
        }

        n_x = x + (-s) * y * powf(2.0f, -(float)i) ;
        n_y = y + s * x * powf(2.0f, -(float)i) ;
        n_z = z + (-s) * 1.0 * atanf(powf(2.0f, -(float)i) ;

        x = n_x;
        y = n_y;
        z = n_z;
}
```

CORDIC Performance



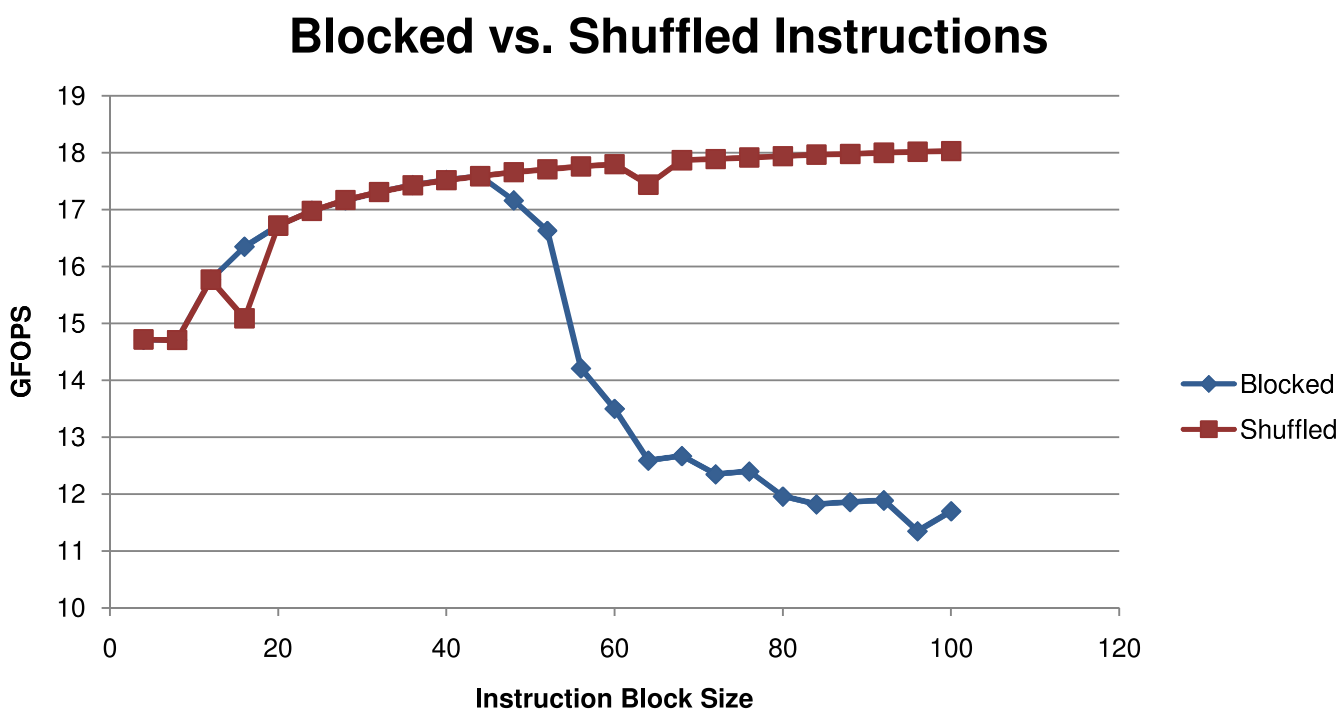
CORDIC Performance Cont.



Software Pipelining

- Out-of-Order scheduling is built around the idea of automatically pipelining code
- Out-of-Order scheduler is effective while multiple independent blocks of code appear in its look ahead window
- Questions:
  - How far ahead does the scheduler look?
  - How effective is “effective”?

Out-of-Order Scheduler



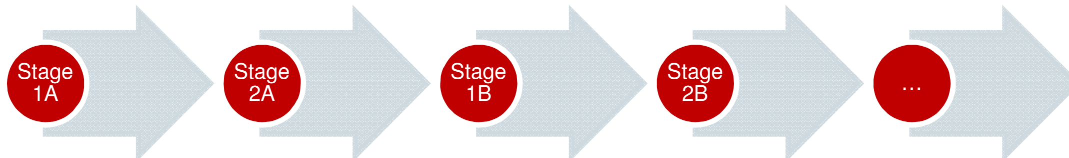
Memory Dependent Tests		
Blocked	Shuffled	Gain
15.10GFLOPS	16.00GFLOPS	5.62%

Software Pipelining

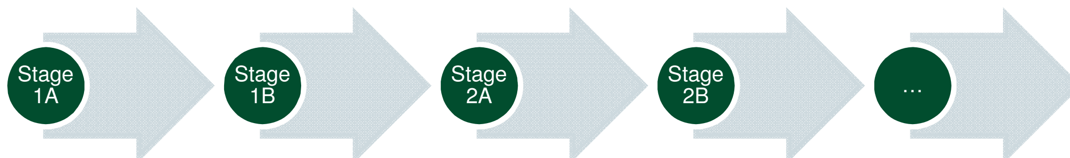
- Scheduler is quite effective as long as instructions are available in its window
- Instructions easily made “available” by two techniques:
  - Unrolling loops
  - Staging loops

Software Pipelining

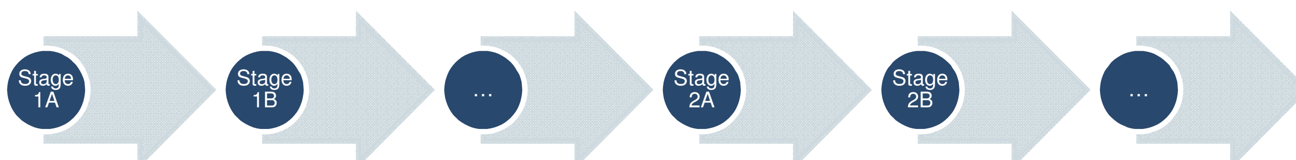
Original



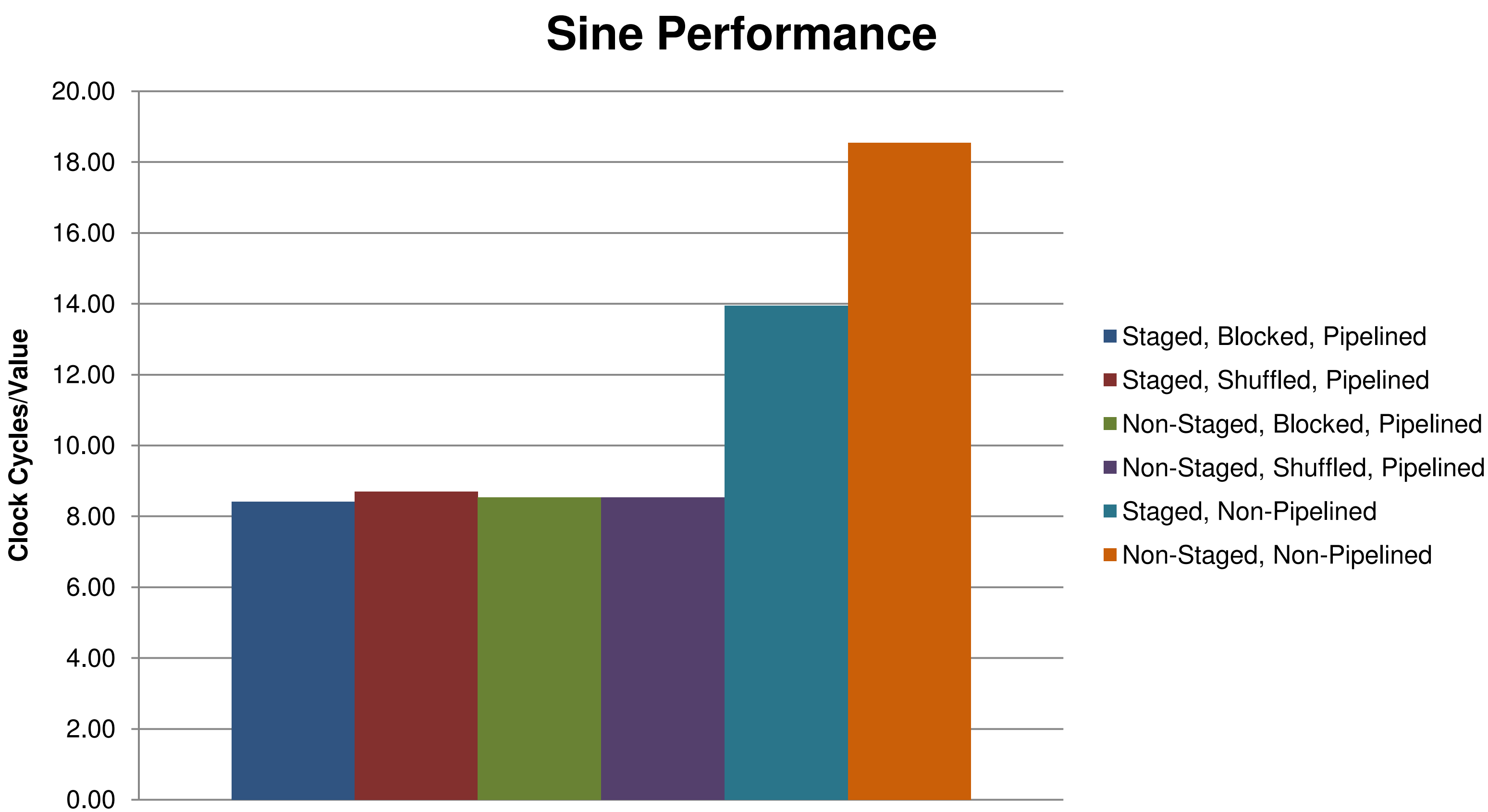
Unrolled



Staged



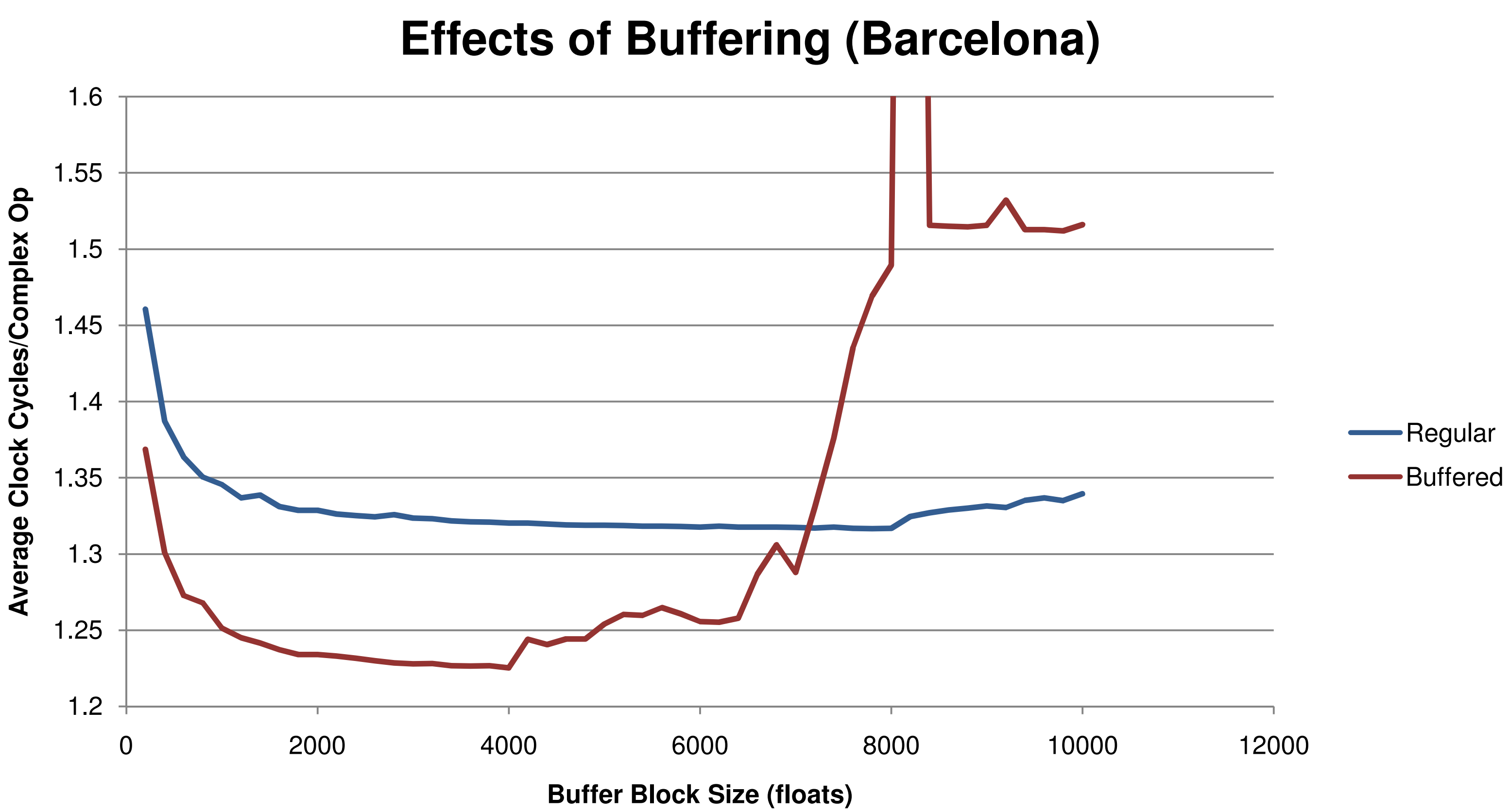
Software Pipelining



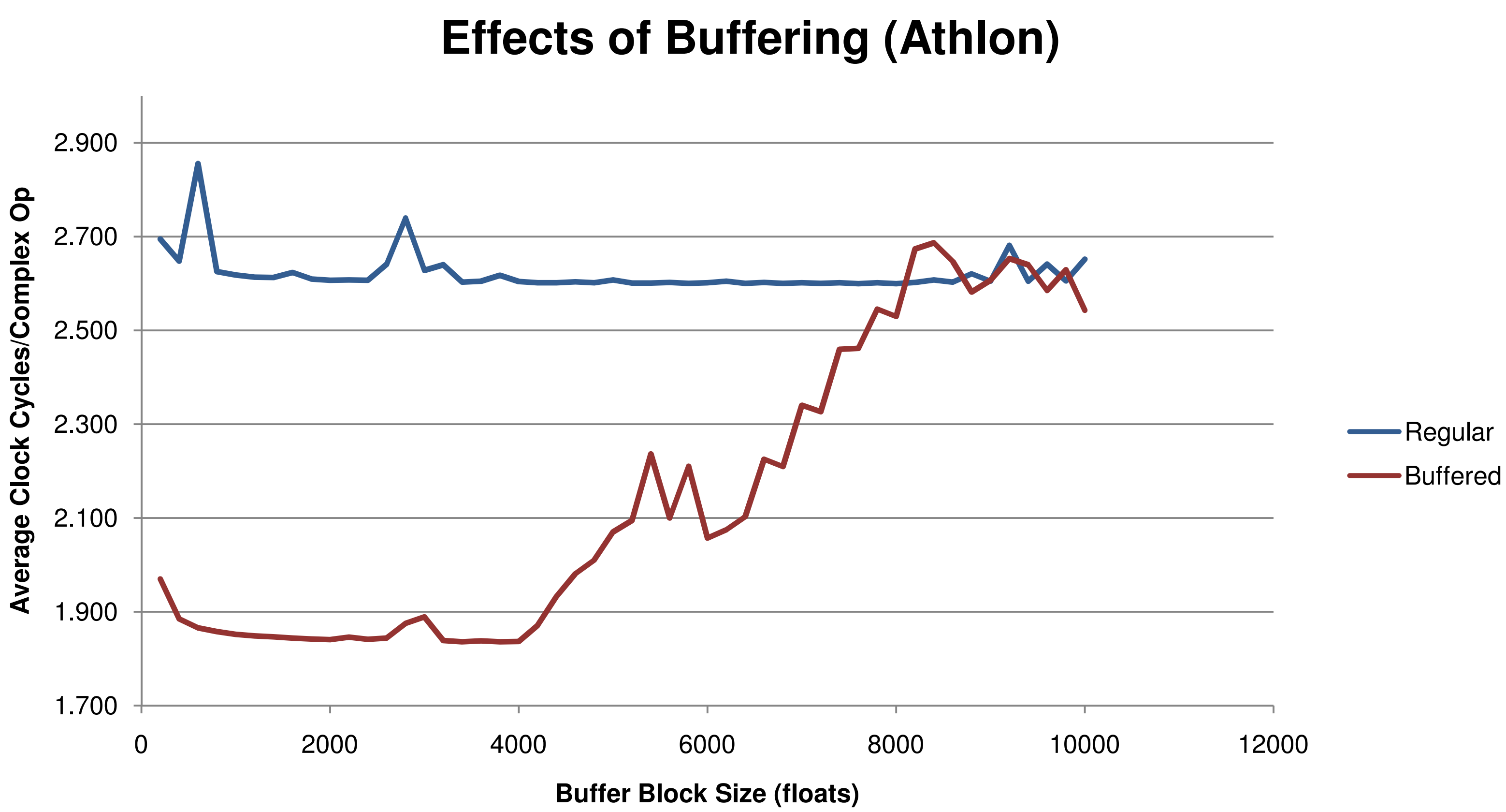
Vector Optimizations

- Some codes pre-shuffle values to make vector operations fit better to SIMD instruction sets
- For multiplication on complex numbers we can trade a dependency and a shuffle for a load

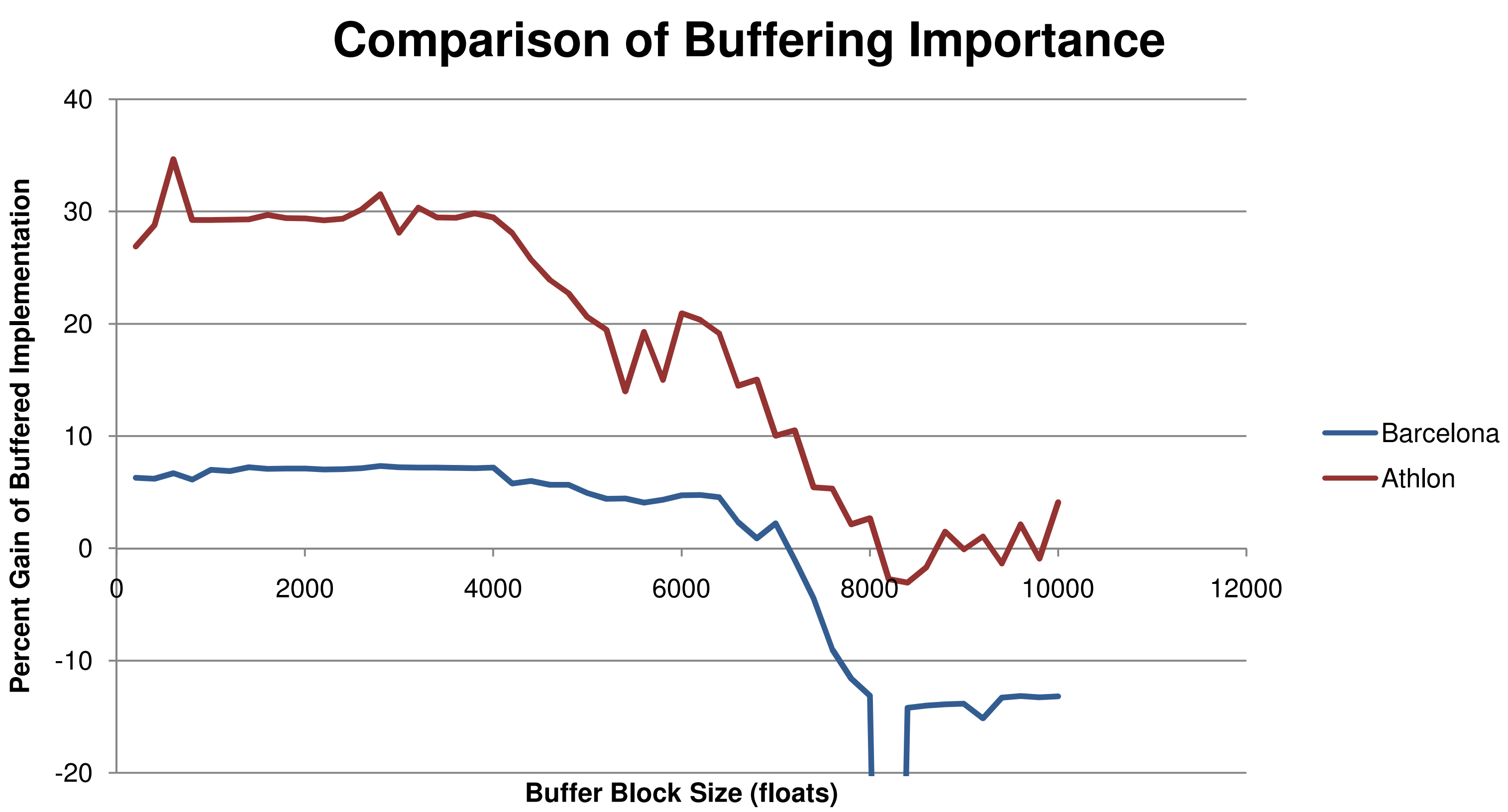
Vector Optimizations



Vector Optimizations



Vector Optimizations



How to Use This Stuff

- All these tests were written in C with the help of the Intel Assembly Intrinsics
  - Unfortunately, the intrinsics do not work well on PGI or Pathscale compilers (they do work in GNU, Intel, and Microsoft ones)
  - Intrinsics, while better than assembly, are clunky at best

How to Use This Stuff

```
for(i = 0; i < ITERS; i++) {
    n_two = _mm_set1_ps(-2.0f);
    m = (__m128)_mm_shuffle_epi32((__m128i)v1, 0x0);
    m = _mm_cmpgt_ps(m, zero);
    n_two = _mm_and_ps(n_two, m);
    n_two = _mm_add_ps(n_two, p_one);
    n_v = (__m128)_mm_shuffle_epi32((__m128i)v1, 0xF1);
    r1 = _mm_load_ps(&lin_lookup[i * 4]);
    n_v = _mm_mul_ps(n_v, r1);
    n_v = _mm_mul_ps(n_two, n_v);
    v1 = _mm_add_ps(v1, n_v);
}
```

Conclusions

- Conditionals
  - Unroll slow conditional loops
  - Use integer comparisons
  - Code for conditional moves
- Pipeline operations explicitly
- Separate staged computations
- Don’t worry about vector operations

Square Root/Reciprocal Time