Exploring the Cell with HPEC Challenge Benchmarks

Sharon M. Sacco, Glenn Schrader, Jeremy Kepner, and Matthew Marzilli

HPEC Conference

21 September 2006

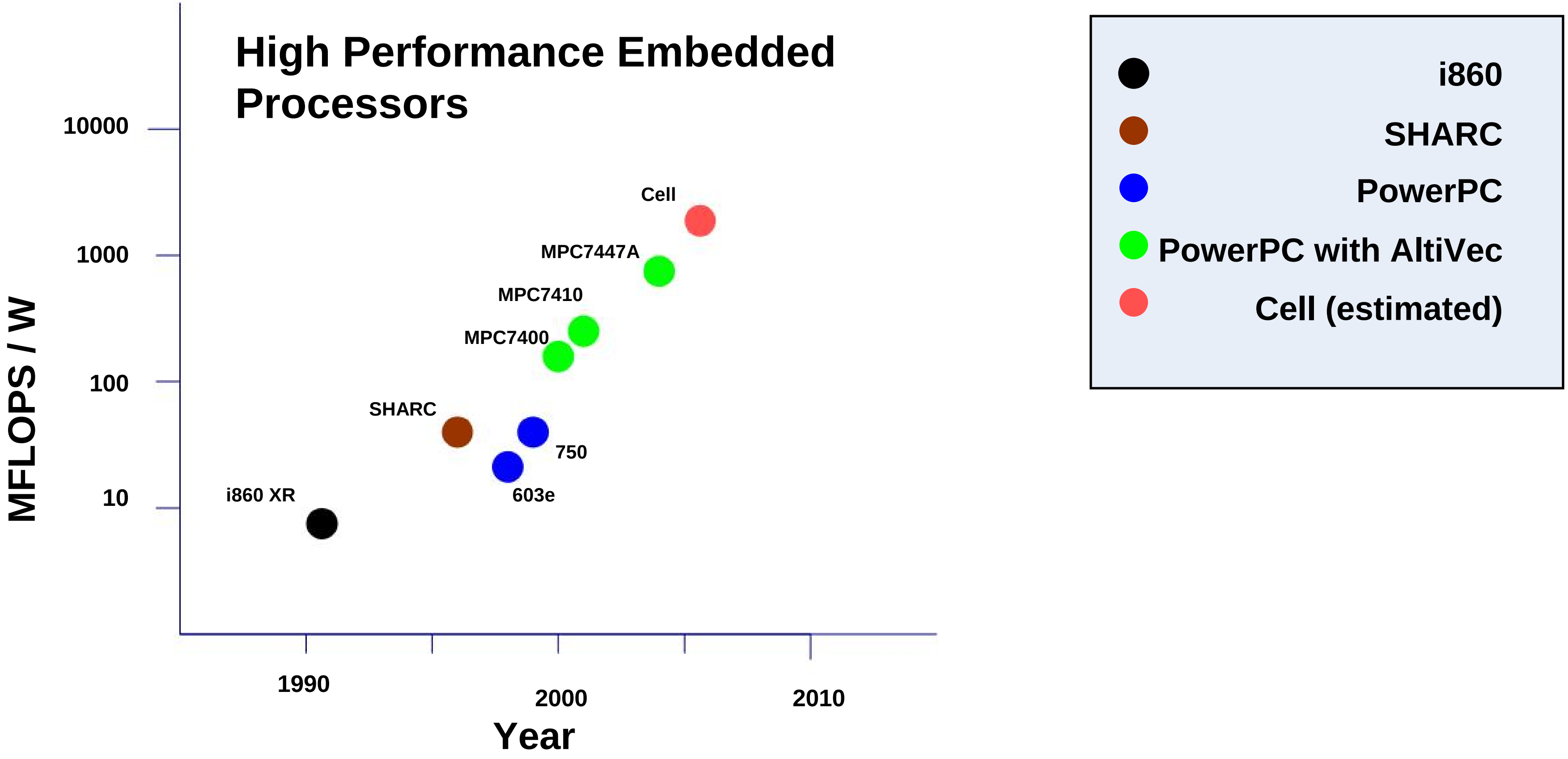
This work is sponsored by AFRL under Air Force contract FA8721-050C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and not necessarily endorse by the United States Government.

Outline

- **Introduction**
- **Approach**
- **Results**
- **Summary**

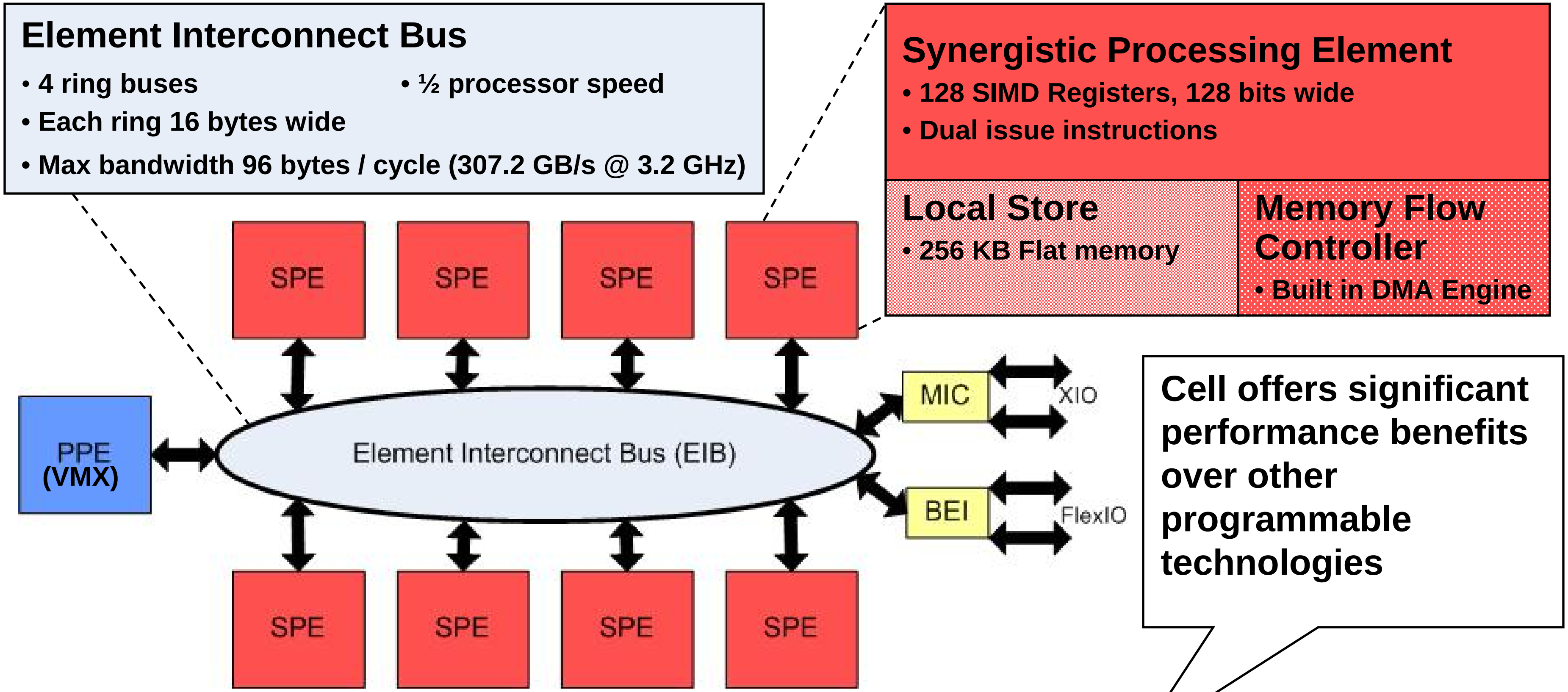
- ***Embedded Processor Evolution***
- ***Cell Features***
- ***Programming Challenges***
- ***Performance Challenge***

Embedded Processor Evolution



- 20 years of exponential growth in FLOPS / W
- Requires switching architectures every ~5 years
- Cell Processor is current high performance architecture

Cell Features



Element Interconnect Bus

- 4 ring buses
- ½ processor speed
- Each ring 16 bytes wide
- Max bandwidth 96 bytes / cycle (307.2 GB/s @ 3.2 GHz)

Synergistic Processing Element

- 128 SIMD Registers, 128 bits wide
- Dual issue instructions

Local Store

- 256 KB Flat memory

Memory Flow Controller

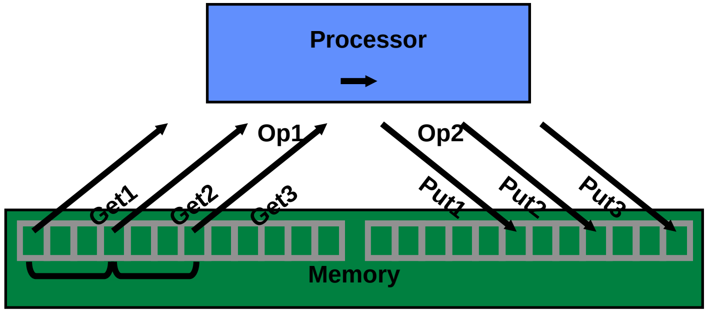
- Built in DMA Engine

Cell offers significant performance benefits over other programmable technologies

- **Overall Performance**
- Peak FLOPS @ 3.2 GHz: 204.8 GFLOPS (single), 14.6 GFLOPS (double)
- Processor to Memory bandwidth: 25.6 GB/s
- Power usage: ~100 W (estimated)
- Cell gives ~2 GFLOPS / W

Programming Challenge

Past: Von Neumann Programming Model



- Great success of Moore’s Law Era
  - Simple user model: get, op, put
  - Many transistors devoted to delivering this model to user
- No longer feasible
  - Need these transistors to improve performance (Intel’s “right turn”)
  - Exposes complex processor topology to user

Future: “Acronym” Processor Programming Model

FPGA: Field Programmable Gate Array

PCA: Polymorphous Computing Architecture

SPE: Synergistic Processing Engine

MTA: Multi-Threaded Architecture

PIM: Processor In Memory

PIN: Processor In Network

PID: Processor In Disk

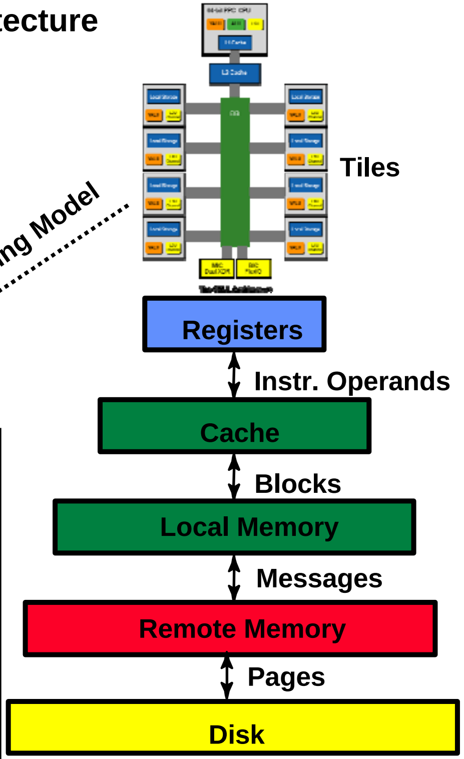
HyperThreading

STI Cell

Vector Processors ...

Cell Programming Model

1. Asymmetric-Thread Runtime
2. Function-Offload
3. Device-Extension
4. Computation-Acceleration
5. Streaming
6. Shared-Memory Multiprocessor
7. User-Mode Thread



- Increased performance at the cost of exposing the full processor topology to the programmer

Performance Challenges

Dual issue instructions

Efficiently partition the application

Keep the pipelines full

Don’t let the cache slow things down

Maximize the use of SIMD Registers

Can the buses be controlled efficiently?

Keep the data flowing

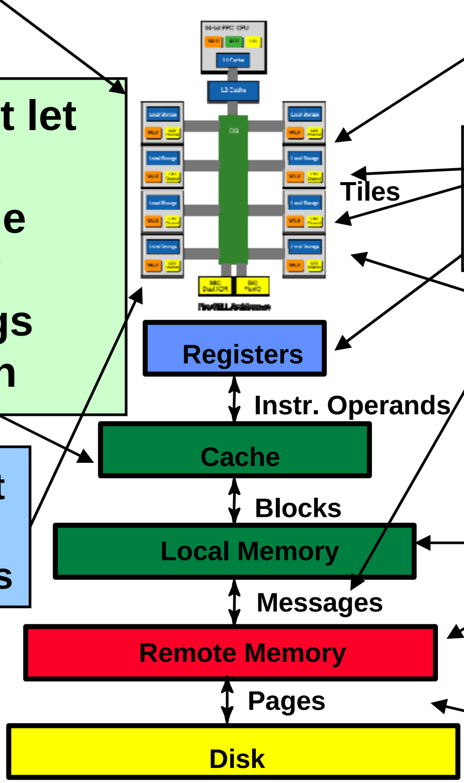
Access memory efficiently

Can the exact processor be selected?

Watch out for race conditions

Cover memory transfers with computations

Can information on disk be preloaded before needed?



- Price of performance is increased programming complexity

Outline

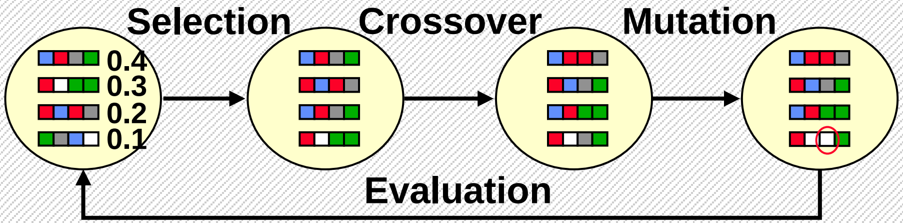
- Introduction
- **Approach**
- Results
- Summary

- ***HPEC Challenge***
- ***Test System***
- *Software Environment*
  - *Time Domain FIR*
  - *Programs*
  - *Parallel Approach*
  - *Octave*
  - *Mercury SAL and MCF*

HPEC Challenge

Information and Knowledge Processing Kernels

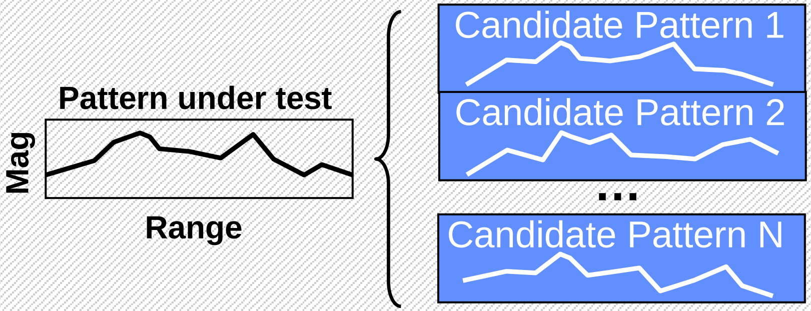
Genetic Algorithm



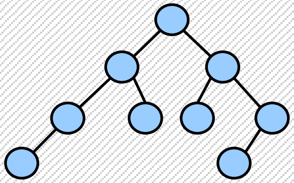
- Evaluate each chromosome
- Select chromosomes for next generation
- Crossover: randomly pair up chromosomes and exchange portions
- Mutation: randomly change each chromosome

Pattern Match

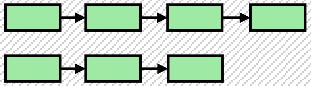
- Compute best match for a pattern out of set of candidate patterns
  - Uses weighted mean-square error



Database Operations



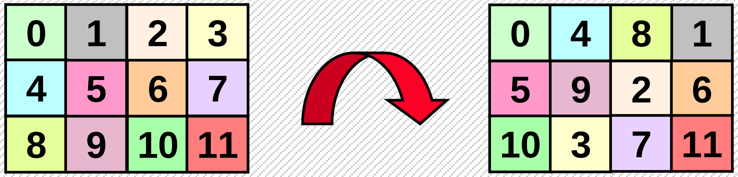
Red-Black Tree Data Structure



Linked List Data Structures

- Three generic database operations:
  - search: find all items in a given range
  - insert: add items to the database
  - delete:remove item from the database

Corner Turn



0	1	2	3
4	5	6	7
8	9	10	11

0	4	8	1
5	9	2	6
10	3	7	11

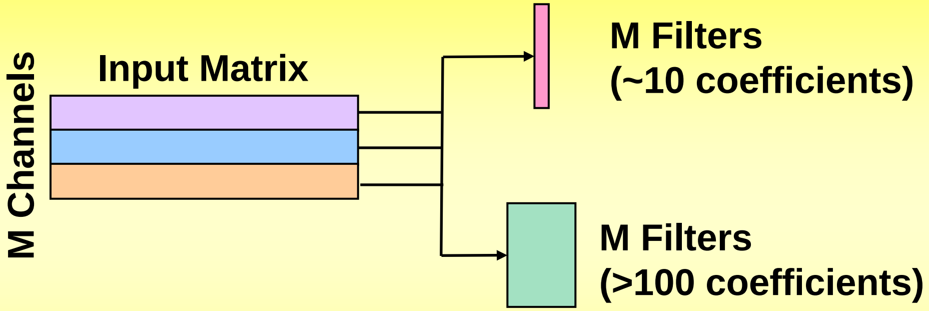
* Numbers denote memory content

- Memory rearrangement of matrix contents
  - Switch from row to column major layout

HPEC Challenge

Signal and Image Processing Kernels

FIR



M Channels

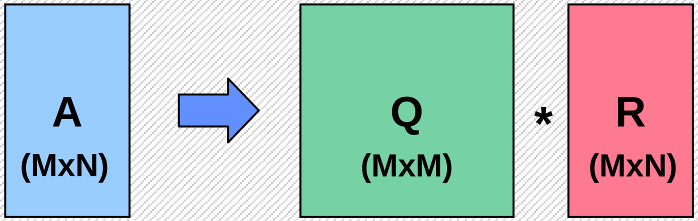
Input Matrix

M Filters (~10 coefficients)

M Filters (>100 coefficients)

- Bank of filters applied to input data
- FIR filters implemented in time and frequency domain

QR

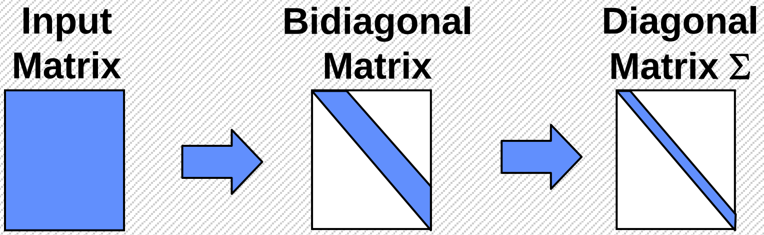


A (MxN)

Q (MxM) * R (MxN)

- Computes the factorization of an input matrix, A=QR
- Implementation uses Fast Givens algorithm

SVD



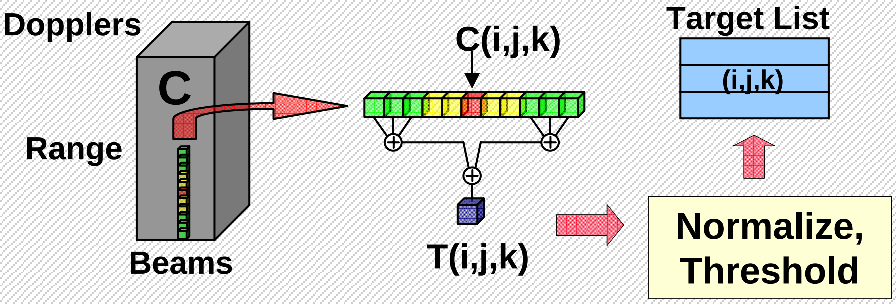
Input Matrix

Bidiagonal Matrix

Diagonal Matrix Σ

- Produces decomposition of an input matrix, $X=U\Sigma V^H$
- Classic Golub-Kahan SVD implementation

CFAR



Dopplers

C

Range

Beams

C(i,j,k)

T(i,j,k)

Target List

(i,j,k)

Normalize, Threshold

- Creates a target list given a data cube
- Calculates normalized power for each cell, thresholds for target detection

Mercury Cell Processor Test System

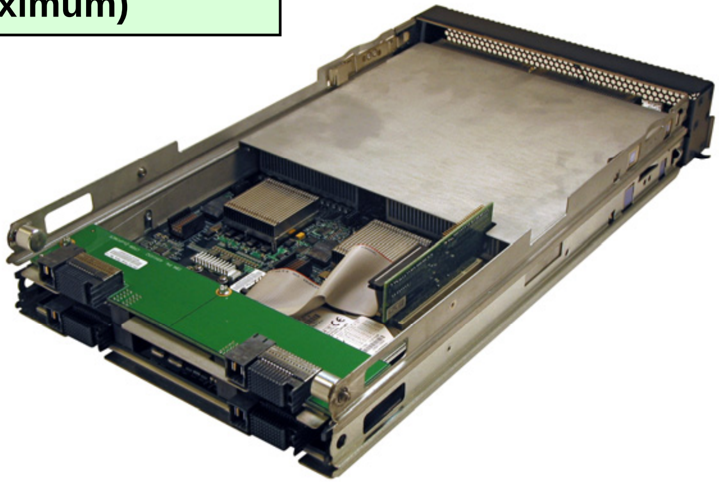
Mercury Cell Processor System

- Single Dual Cell Blade
  - Native tool chain
  - Two 2.4 GHz Cells running in SMP mode
  - Terra Soft Yellow Dog Linux 2.6.14
- Received 03/21/06
  - booted & running same day
  - integrated/w LL network < 1 wk
  - Octave (Matlab clone) running
  - Parallel VSIPL++ compiled

•Each Cell has 153.6 GFLOPS (single precision)
– 307.2 for system @ 2.4 GHz (maximum)

Software includes:

- IBM Software Development Kit (SDK)
  - Includes example programs
- Mercury Software Tools
  - MultiCore Framework (MCF)
  - Scientific Algorithms Library (SAL)
  - Trace Analysis Tool and Library (TATL)

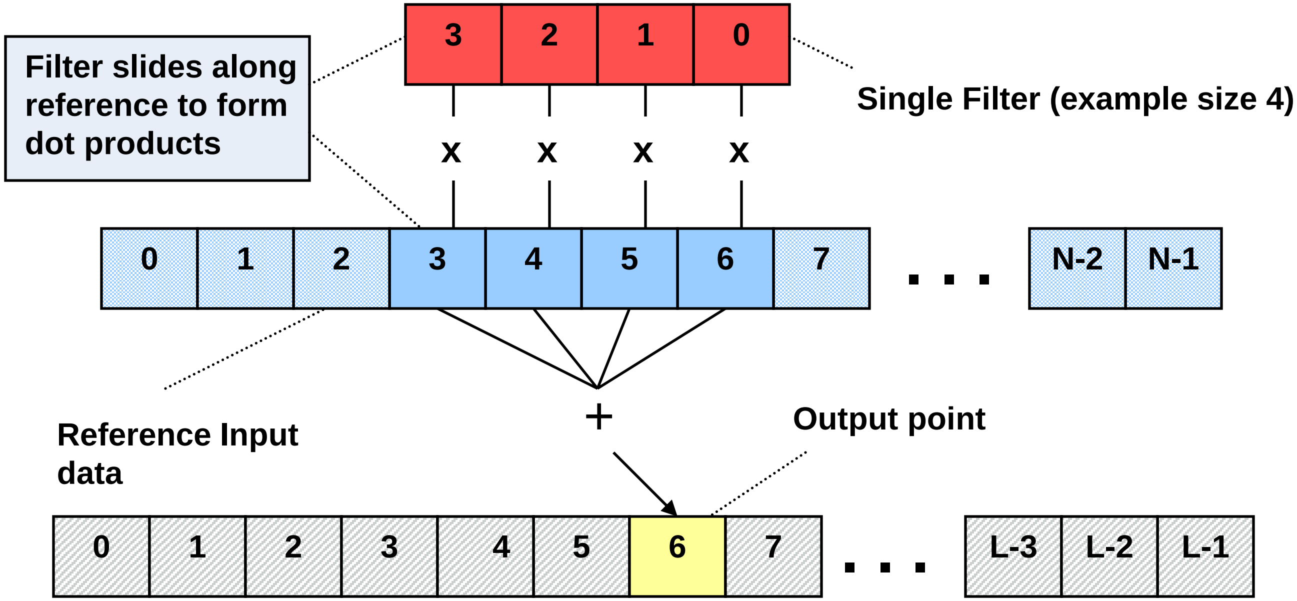


Outline

- Introduction
- **Approach**
- Results
- Summary

- *HPEC Challenge*
- *Test System*
- ***Software Environment***
  - ***Time Domain FIR***
  - ***Programs***
  - ***Parallel Approach***
  - *Octave*
  - *Mercury SAL and MCF*

Time Domain FIR Algorithm



Filter slides along reference to form dot products

3 2 1 0

Single Filter (example size 4)

x x x x

0 1 2 3 4 5 6 7 . . . N-2 N-1

+

Reference Input data

Output point

0 1 2 3 4 5 6 7 . . . L-3 L-2 L-1

- Number of Operations:

K – Filter size

N – Input size

M – Number of filters

Total FOPs: ~ 8 x M x N x K

- Output Size:

L = N + K - 1

- TDFIR uses complex data
- TDFIR uses a bank of filters
  - Each filter is used in a tapered convolution
  - A convolution is a series of dot products

HPEC Challenge Parameters TDFIR

Set	K	N	M
1	128	4096	64
2	12	1024	20

•FIR is one of the best ways to demonstrate FLOPS

Reference C implementation

- Computations take 2 lines
- Mostly loop control, pointers, and initialization
- Output initialization assumed
- SPE needs split complex
    - Separate real and imaginary vectors

Reference C FIR is easy to understand

```
for (i = K; i > 0; i--){

   /* Set accumulators and pointers for dot product
                  for output point */
   r1 = Rin;
   r2 = Iin;
   o1 = Rout;
   o2 = Iout;

   /* calculate contributions from a single kernel point */
   for (j = 0; j < N; j++){

     *o1 += *k1 * *r1 - *k2 * *r2;
     *o2 += *k2 * *r1 + *k1 * *r2;

     r1++; r2++; o1++; o2++;
   }

   /* update input pointers */
   k1++; k2++;
   Rout++;
   Iout++;
 }
```

C with SIMD Extensions

- Inner loop contributes to 4 output points per pass
- SIMD registers in use
- Shuffling of values in registers is a requirement
  - Compilers are unlikely to recognize this type of code
- Can rival assembly code with more effort

- SIMD C extensions increase code complexity
  - Hardware needs consideration

```
/* load reference data and shift  */
    ir0 = *Rin++;
    ii0 = *Iin++;
    ir1 = (vector float) spu_shuffle(irOld, ir0, shift1);
    ii1 = (vector float) spu_shuffle(iiOld, ii0, shift1);
    ir2 = (vector float) spu_shuffle(irOld, ir0, shift2);
    ii2 = (vector float) spu_shuffle(iiOld, ii0, shift2);
    ir3 = (vector float) spu_shuffle(irOld, ir0, shift3);
    ii3 = (vector float) spu_shuffle(iiOld, ii0, shift3);

    Rtemp = kr0 * ir0 + Rtemp;      Itemp = kr0 * ii0 + Itemp;
    Rtemp = -(ki0 * ii0 - Rtemp);   Itemp = ki0 * ir0 + Itemp;

    Rtemp = kr1 * ir1 + Rtemp;      Itemp = kr1 * ii1 + Itemp;
    Rtemp = -(ki1 * ii1 - Rtemp);   Itemp = ki1 * ir1 + Itemp;

    Rtemp = kr2 * ir2 + Rtemp;      Itemp = kr2 * ii2 + Itemp;
    Rtemp = -(ki2 * ii2 - Rtemp);    Itemp = ki2 * ir2 + Itemp;

    Rtemp = kr3 * ir3 + Rtemp;      Itemp = kr3 * ii3 + Itemp;
    Rtemp = -(ki3 * ii3 - Rtemp);   Itemp = ki3 * ir3 + Itemp;

    *Rout++ = Rtemp;   *Iout++ = Itemp;

    irOld = ir0;     iiOld = ii0;          /* update old values    */
```

Contents of inner loop of convolution

SPE Assembly Version

Dual issue instructions

Integer and bitwise instructions compete with floating point for dual issue

Padding with occasional “no operations” keeps the performance going

Easy to end dual issue instructions

Pointer updates and loop control affect inner loop cycle count

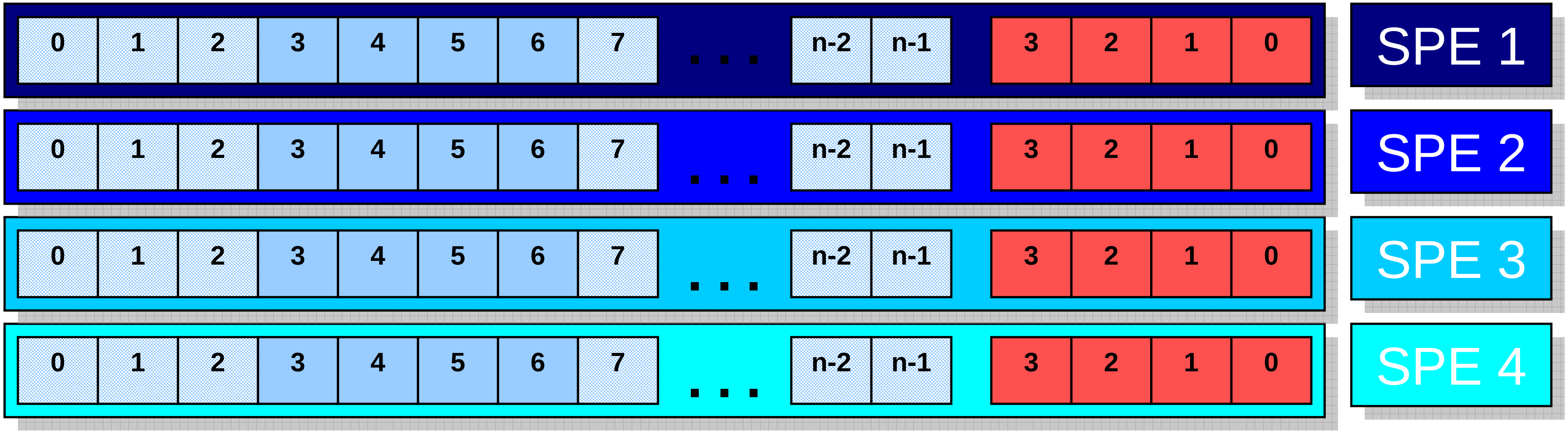
```
/* Fourth kernel point contribution  */
 fma $50,$30,$66,$50        /* reout''' + rein * rek [4i+16j+3]-[4i+16j+6]      */
 lqd $45,32($15)            /* load imin[16j+24]-[16j+27]                       */
 fma $51,$31,$66,$51        /* imout''' + rein * imk [4i+16j+3]-[4i+16j+6]      */
 shufb $66,$40,$42,$16      /* rein[4i+16j+17]-[4i+16j+20] in $66               */
 fma $52,$30,$68,$52        /* reout''' + rein * rek [4i+16j+7]-[4i+16j+10]     */
 lqd $47,48($15)            /* load imin[16j+28]-[16j+31]                       */
 fma $53,$31,$68,$53        /* imout''' + rein * imk [4i+16j+7]-[4i+16j+10]     */
 shufb $68,$42,$44,$16      /* rein[4i+16j+21]-[4i+16j+24] in $68               */
 and $43,$43,$19            /* clear $43 for taper if necessary                 */
 lnop
 fma $54,$30,$70,$54        /* reout''' + rein * rek [4i+16j+11]-[4i+16j+14]    */
 lqd $49,64($15)            /* load imin[4i+16j+32]-[4i+16j+35]                 */
 fma $55,$31,$70,$55        /* imout''' + rein * imk [4i+16j+11]-[4i+16j+14]    */
 shufb $70,$44,$46,$16      /* rein[4i+16j+25]-[4i+16j+28] in $70               */
 fma $56,$30,$72,$56        /* reout''' + rein * rek [4i+16j+15]-[4i+16j+18]    */
 ai $14,$14,64              /* update rein address                              */
 fma $57,$31,$72,$57        /* imout''' + rein * imk [4i+16j+15]-[4i+16j+18]    */
 shufb $72,$46,$48,$16      /* rein[4i+16j+29]-[4i+16j+32] in $72               */
```

Code fragment from inner loop (10 cycles)

- Inner loop operates on 16 output values with 4 kernel values
- 78% dual issue cycles in inner loop with 4 nops
- 74 registers used

- **High performance demands software to leverage hardware**

Parallel Approach Time Domain FIR


0 1 2 3 4 5 6 7 . . . n-2 n-1
3 2 1 0
SPE 1
0 1 2 3 4 5 6 7 . . . n-2 n-1
3 2 1 0
SPE 2
0 1 2 3 4 5 6 7 . . . n-2 n-1
3 2 1 0
SPE 3
0 1 2 3 4 5 6 7 . . . n-2 n-1
3 2 1 0
SPE 4

- **HPEC Challenge Benchmark TDFIR is a series of independent convolutions**
  - **"Embarrassingly" parallel problem is a good place to start**
  - **Independent convolutions are divided among the processors**
  - **Computation of one convolution can be overlapped with DMAs from others**

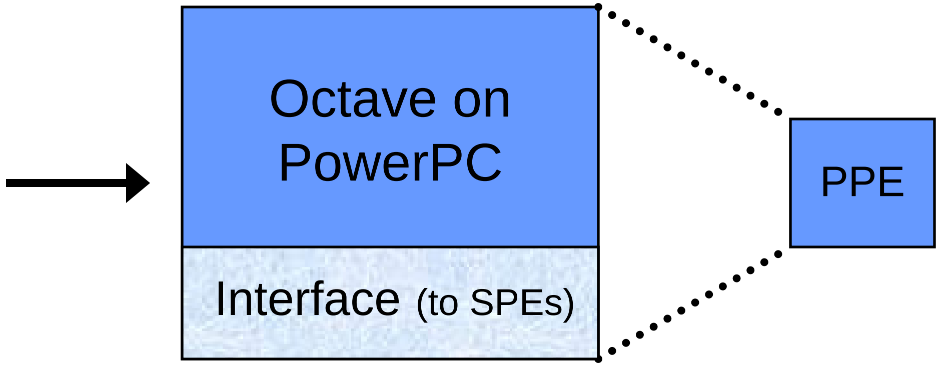
Outline

- Introduction
- **Approach**
- Results
- Summary

- *HPEC Challenge*
- *Test System*
- ***Software Environment***
  - *Time Domain FIR*
  - *Programs*
  - *Parallel Approach*
  - ***Octave***
  - ***Mercury SAL and MCF***

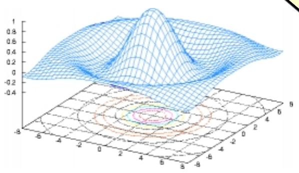
Communication with Octave

```
A = rand_data()
B = rand_data()

C = call_cell (A,B)

verify(C1)
          …
```


Octave on PowerPC

Interface (to SPEs)

PPE



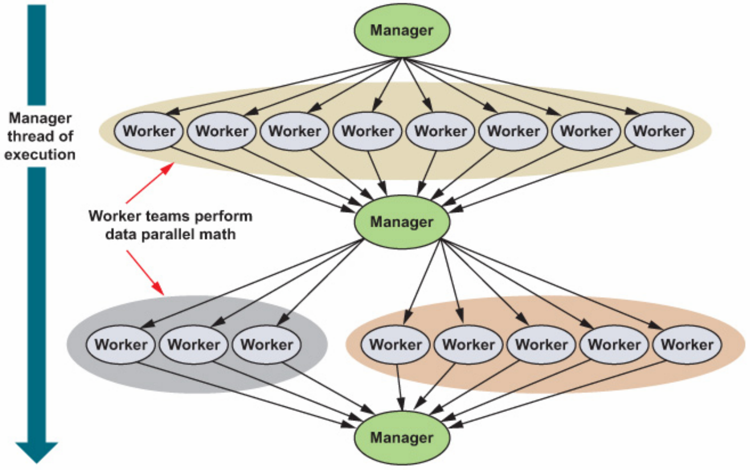
- Octave lets MATLAB code to run on Cell
  - Ideal for rapid prototyping
- PowerPC makes complex programs easy to build

Octave is free software:
www.octave.org

- Easy to generate data, verify output, and estimate performance
- MATLAB Executable (MEX) or Octave Native Interface (ONI) provide interface to C code to run SPEs from the PPE

Mercury SAL and MCF

- **Scientific Algorithms Library (SAL)** is an FPS based library available on most Mercury products
  - Program portability within Mercury products
  - Common signal processing algorithms



- SAL has over 100 functions optimized for single SPE
  - FFT (1D, multiple)
  - Convolution (real, complex)
  - Matrix multiply
  - Basic arithmetic
  - Trigonometric and transcendental
  - Transpose

- **MultiCore Frameworks (MCF)** manages multi-SPE programming
  - Function offload engine model
  - Stripmining
  - Intraprocessor communications
  - Overlays
  - Profiling

- Hand code TDFIR once for programming experience
  - Rely on vendor SPE math libraries and kernels for productivity
- Programming DMA is as hard as programming 1 SPE

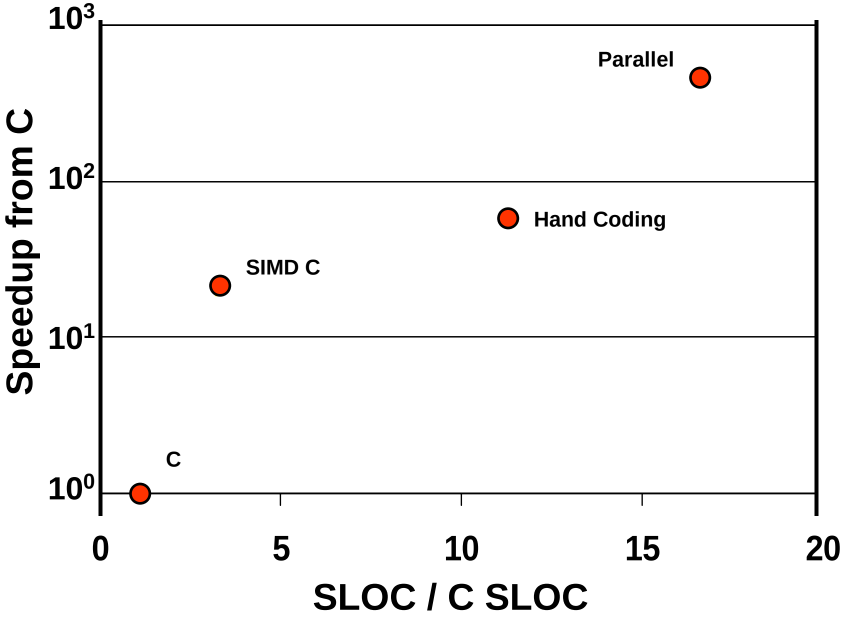
Outline

- Introduction
- Approach
- **Results**
  - ***SLOCs and Coding Effort***
  - ***Performance***
  - ***Overhead***
  - ***HPEC Challenge Results***
- Summary

SLOCs and Coding Effort

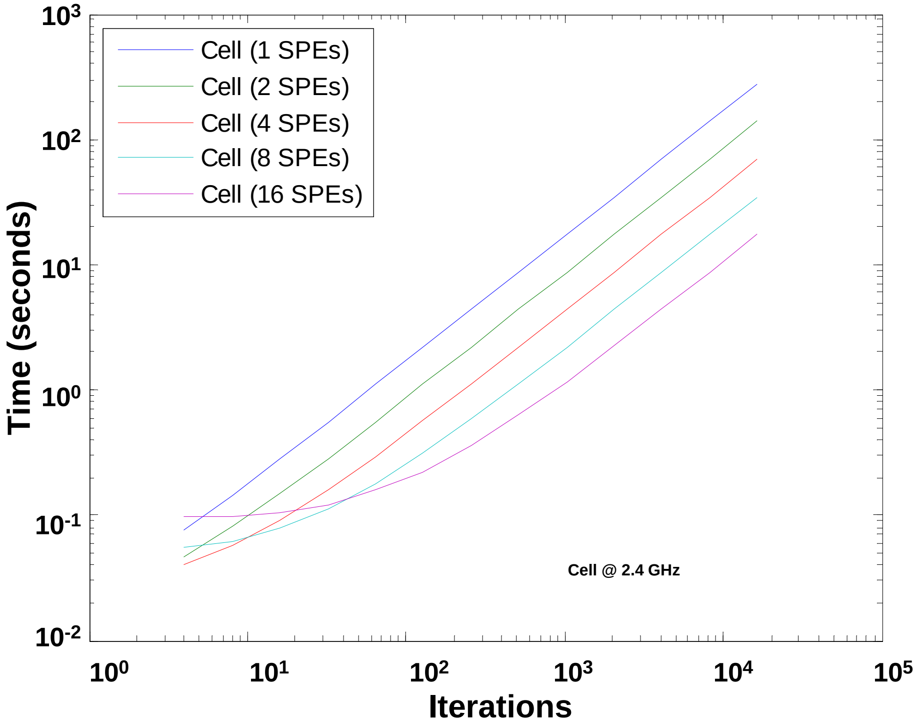
	C	SIMD C	Hand Coding	Parallel (8 SPE)
Lines of Code	33	110	371	546
Design Time	Minute	Hour	Hour	-
Coding Time	Minute	Hour	Day	-
Debug Time	Minute	Minute	Day	-
Performance Efficiency (1 SPE)	0.014	0.27	0.88	0.82
GFLOPS @ 2.4 GHz	0.27	5.2	17	126

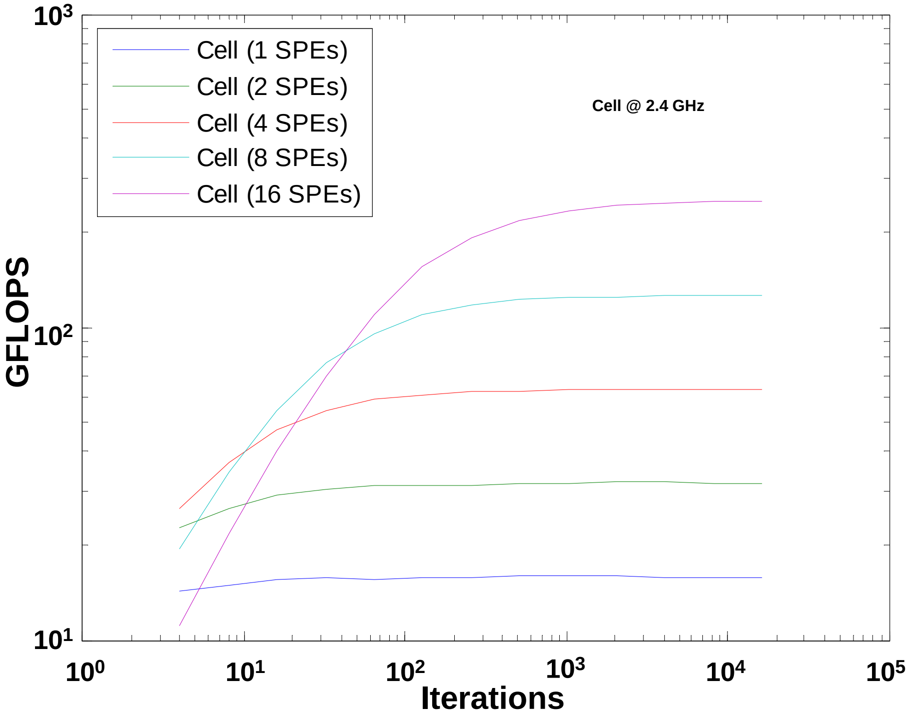
Software Lines of Code (SLOC) and Performance for TDFIR



- **Clear tradeoff between performance and effort**
  - C code simple, poor performance
  - SIMD C, more complex to code, reasonable performance
  - Hand coding, very complex, excellent performance

Performance Time Domain FIR





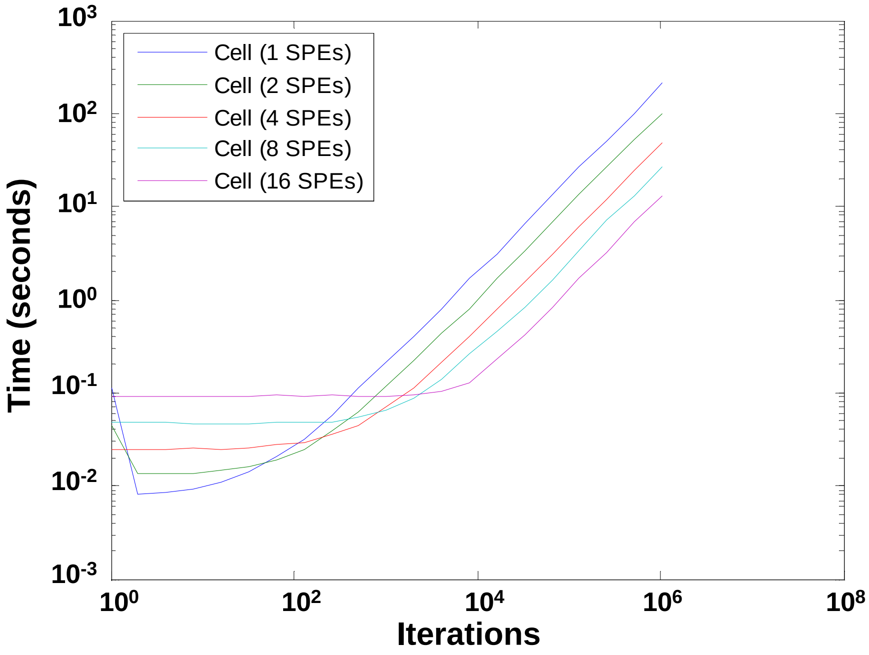
Set 1 has a bank of 64 size 128 filters with size 4096 input vectors

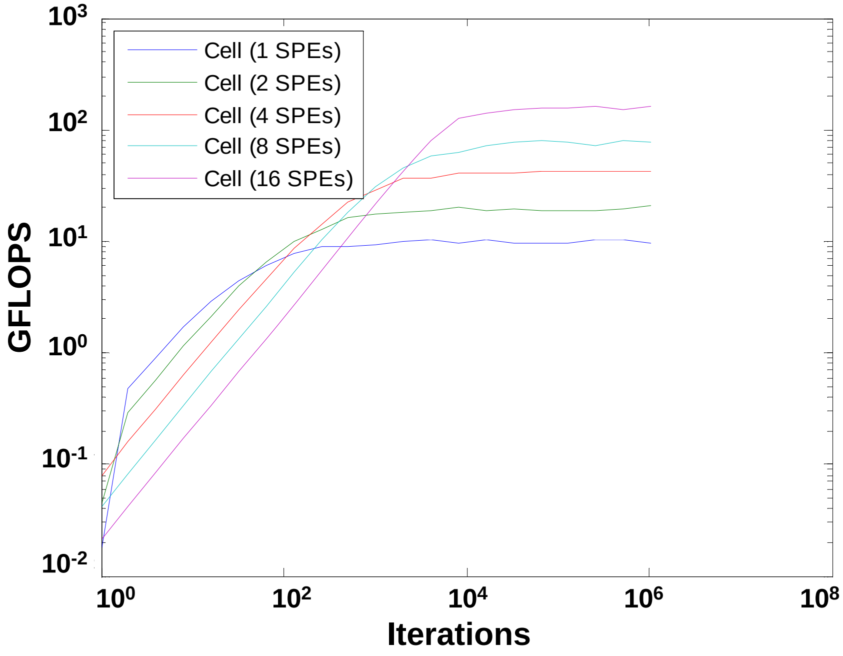
- **Octave runs TDFIR in a loop**
  - **Averages out overhead**
  - **Applications typically run convolutions many times**

Maximum GFLOPS for TDFIR #1

# SPE	1	2	4	8	16
GFLOPS	16	32	63	126	253

Performance Time Domain FIR





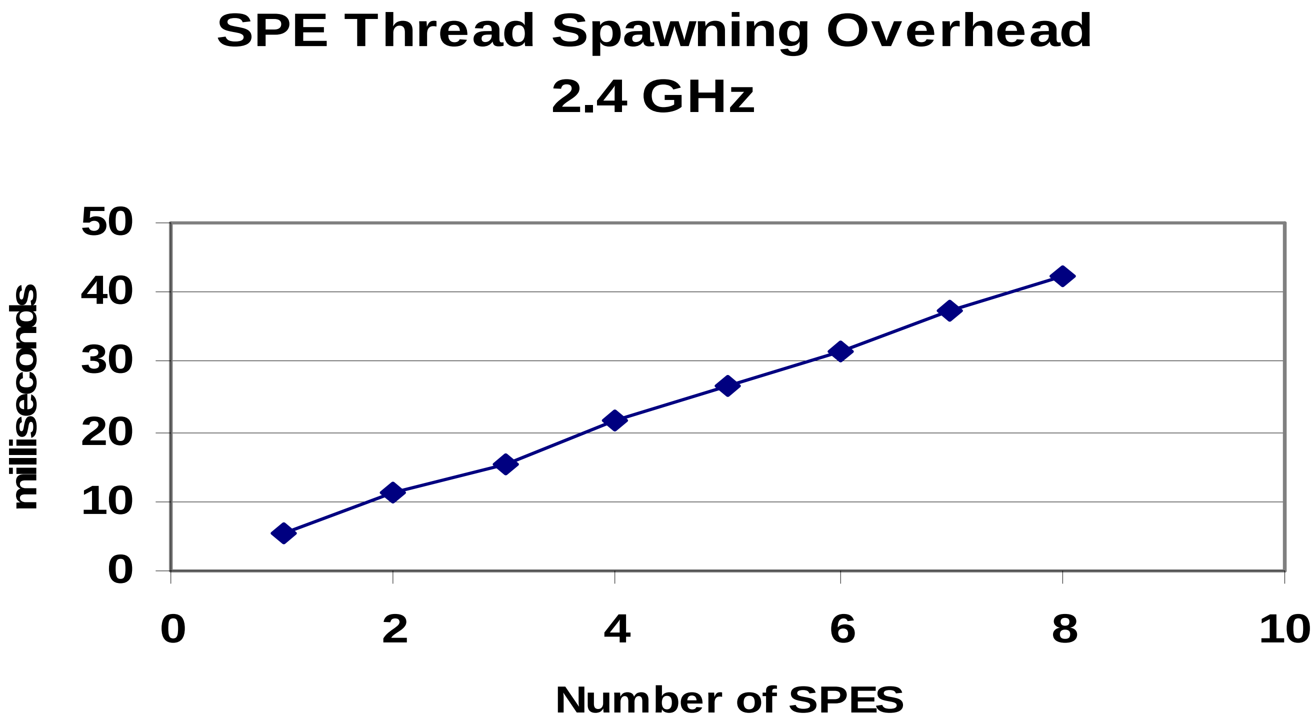
Set 2 has a bank of 20 size 12 filters with size 1024 input vectors

- **TDFIR set 2 scales well with the number of processors**
  - **Runs are less stable than set 1**

GFLOPS for TDFIR #2

# SPE	1	2	4	8	16
GFLOPS	10	21	44	85	185

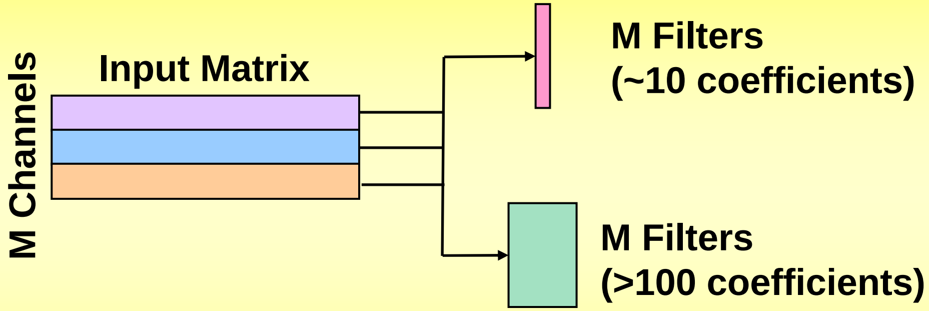
Overhead



- **Thread spawn takes ~ 5.3 ms / SPE**
  - **Minimize thread spawns**
  - **Use middleware that avoids thread spawns**

HPEC Challenge Results

FIR



- Bank of filters applied to input data
- FIR filters implemented in time and frequency domain

- Hand coding lets Cell show performance
- Time domain FIR scales well on Cell
  - Set 1 produces very stable results
  - Set 2 has noticeable variations run to run
    - Does not depend on total time run
    - Variation can be several microseconds

HPEC Challenge TDFIR on Cell @2.4 GHz

# SPE	Set 1	Set 2
1	16.8 ms	190 μs
2	8.50 ms	96 μs
4	4.25 ms	45 μs
8	2.12 ms	23 μs
16	1.06 ms	12 μs

HPEC Challenge Parameters TDFIR

Set	K	N	M
1	128	4096	64
2	12	1024	20

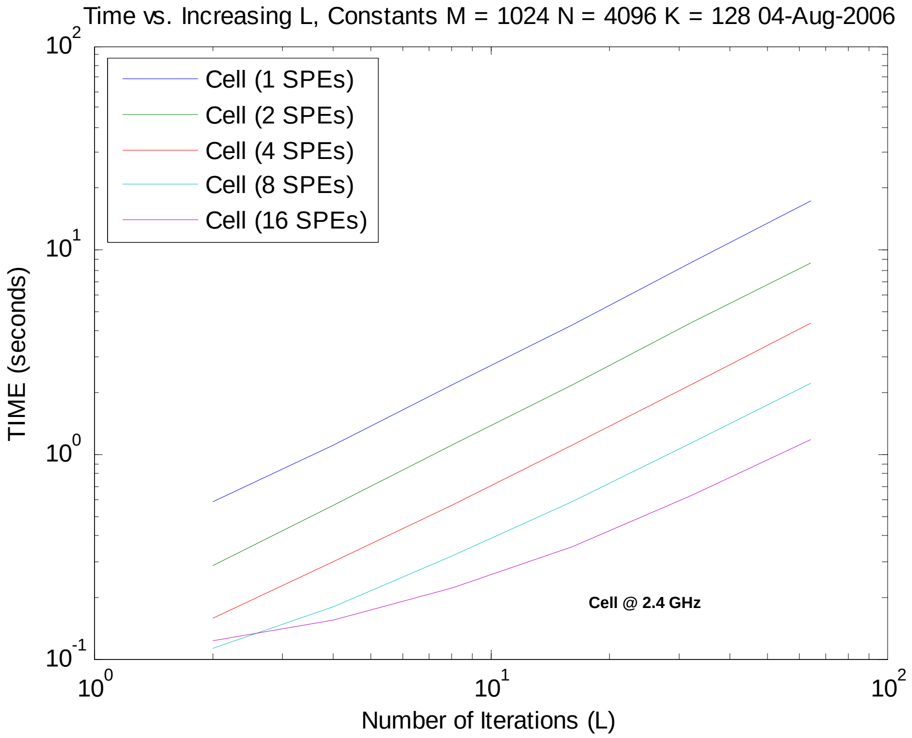
Summary

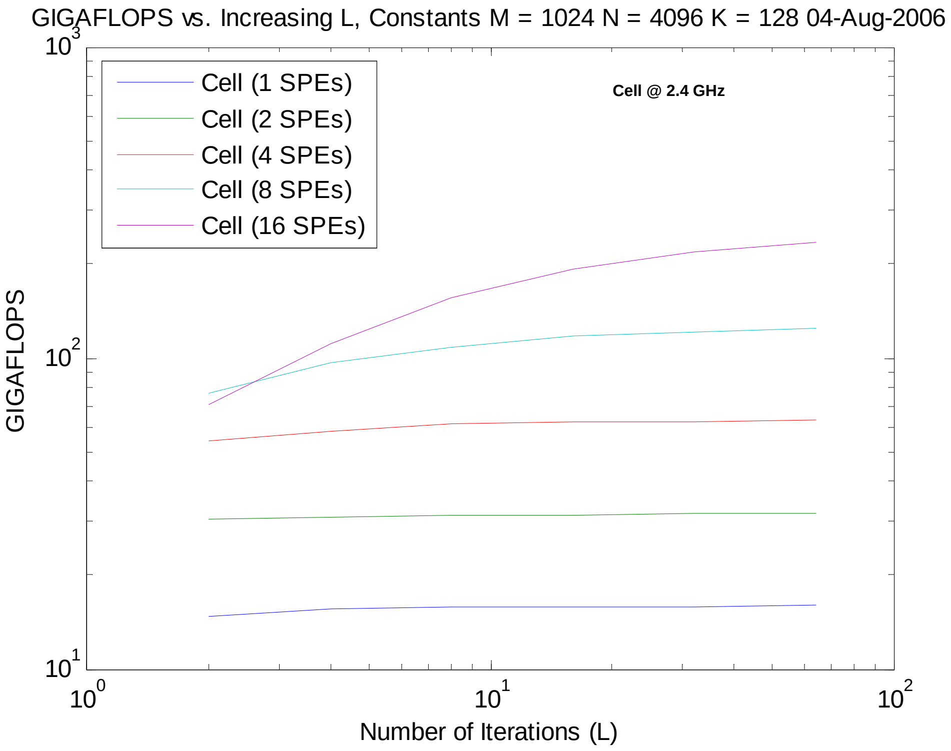
- Good performance has been shown with HPEC Challenge Time Domain FIR on Cell
  - Achieved 80 - 90% performance (253 GFLOPS)
  - Thread spawning overhead (5.3 ms / SPE) should be minimized. Mercury’s MCF is a good alternative.
  - Coding for the SIMD registers give substantial performance improvement over standard C code.
- Future work will expand work with HPEC Challenge Benchmark kernels
  - Frequency Domain FIR will be the next target
  - Need to explore less “embarrassingly” parallel code

Backup Slides

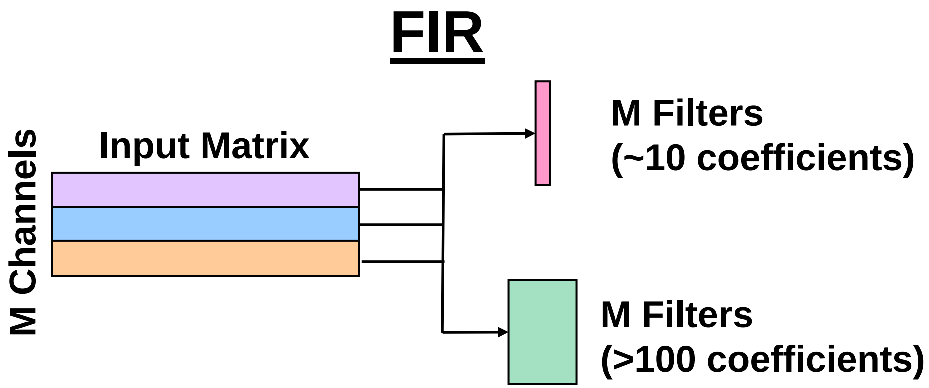
Performance Time Domain FIR

Larger Number of Filters (M) with increasing iterations (L)





HPEC Challenge Results



- Bank of filters applied to input data
- FIR filters implemented in time and frequency domain

- Hand coding lets Cell show performance
- Time domain FIR scales well on Cell
  - Set 1 produces very stable results
  - Set 2 has noticeable variations run to run
    - Does not depend on total time run
    - Variation can be several microseconds

HPEC Challenge TDFIR on Cell @2.4 GHz

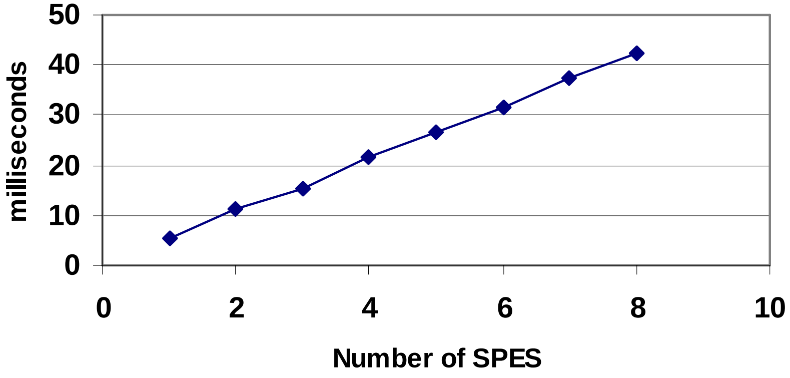
# SPE	Set 1 GFLOPS	Set 2 GFLOPS
1	16.0	10
2	31.6	21
4	63.2	44
8	126	85
16	253	155

HPEC Challenge Parameters TDFIR

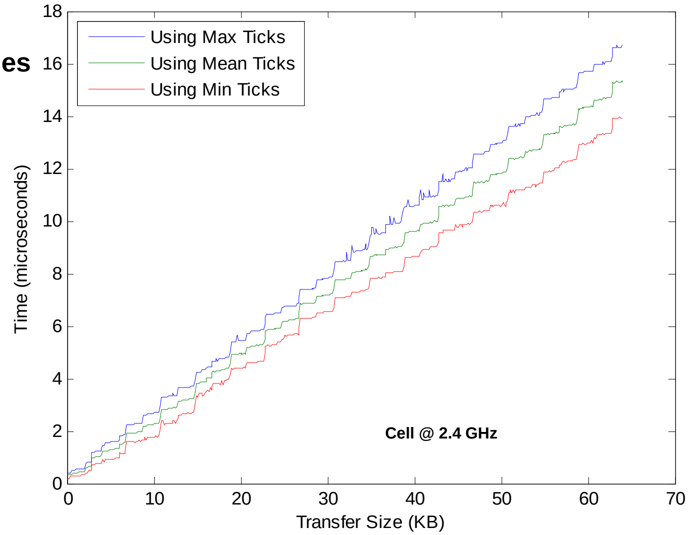
Set	k	n	nf
1	128	4096	64
2	12	1024	20

Overhead

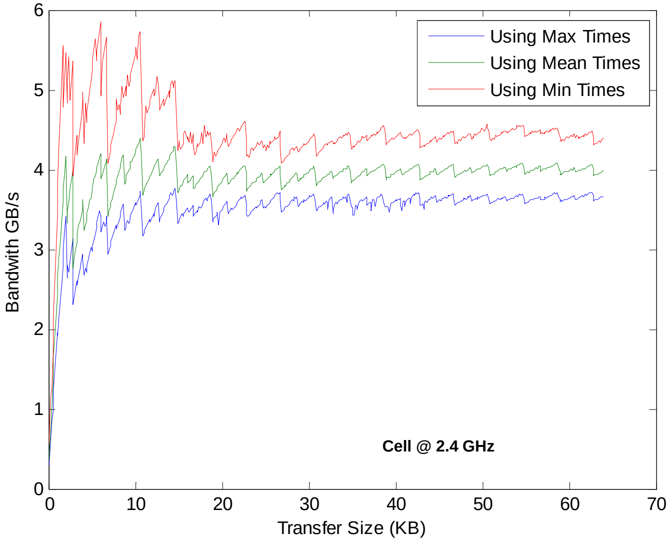
SPE Thread Spawning Overhead
2.4 GHz



DMA Transfer Times
(1 SPE)



DMA Bandwidth
(1 SPE)



- **Thread spawn takes ~ 5.3 ms / SPE**
- **Single DMA take microseconds**
  - **Max transfer is 16 kB**
  - **Larger transfers use DMA lists**
  - **Single SPE does not saturate bandwidth to memory**