A High-Level Signal Processing Library for Multicore Processors

Sharon Sacco, Nadya Bliss, Ryan Haney, Jeremy Kepner, Hahn Kim, Sanjeev Mohindra, Glenn Schrader and Edward Rutledge

This work is sponsored by the Department of the Air Force under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government

Outline

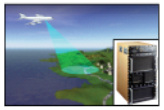
- **Overview**
- HPEC Challenge Benchmarks
- Parallel Vector Tile Optimizing Library
- Summary

Embedded Digital Systems

Rapid System Prototyping

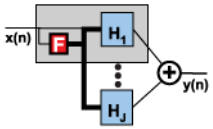
Widebody Airborne Sensor Platform

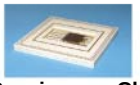
Greenbank



Triple Canopy Foliage Penetration

Advanced Hardware Implementations





Receiver on-Chip

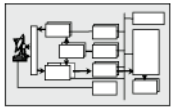
Very Large Scale Integration/ Field Programmable Gate Array Hybrids

Next-Generation Warfighting Vision

Open System Technologies

High Performance Embedded Computing— Software Initiative

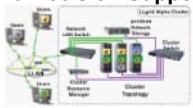
Parallel Matlab



Next Gen Radar Open Systems Architecture

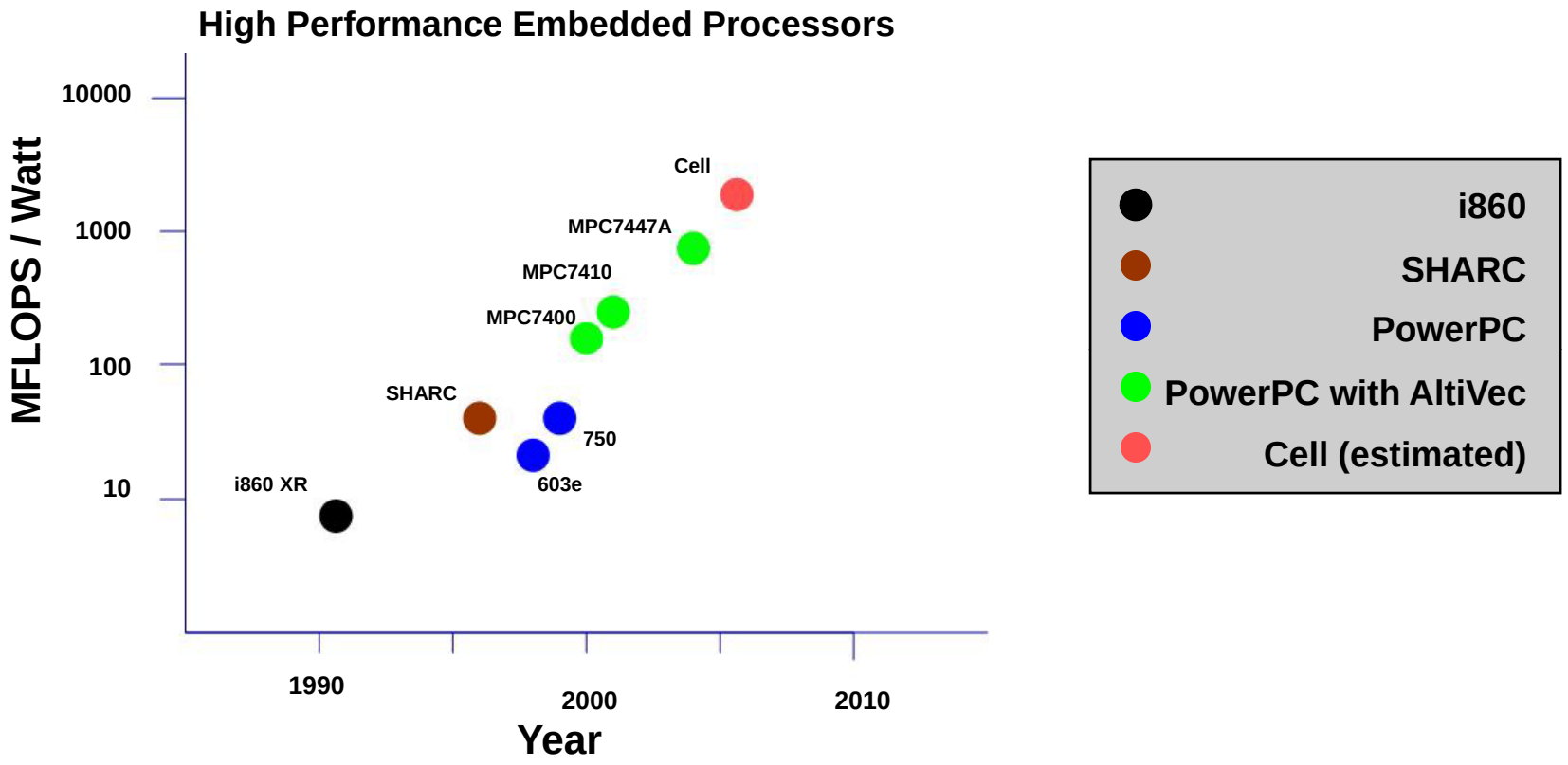
Network & Decision Support Initiatives

Integrated Sensing and Decision Support



LLGrid

Embedded Processor Evolution



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

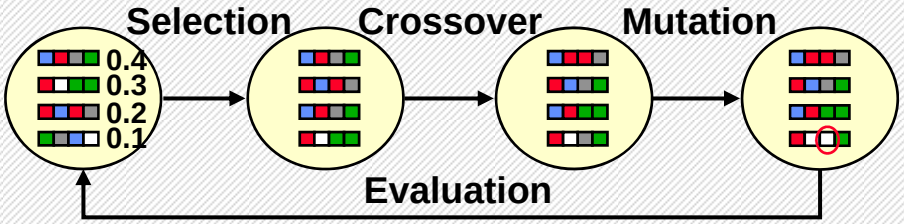
Outline

- Overview
- **HPEC Challenge Benchmarks**
  - ***Time-Domain FIR Filter***
  - ***Results***
- Parallel Vector Tile Optimizing Library
- Summary

HPEC Challenge

Information and Knowledge Processing Kernels

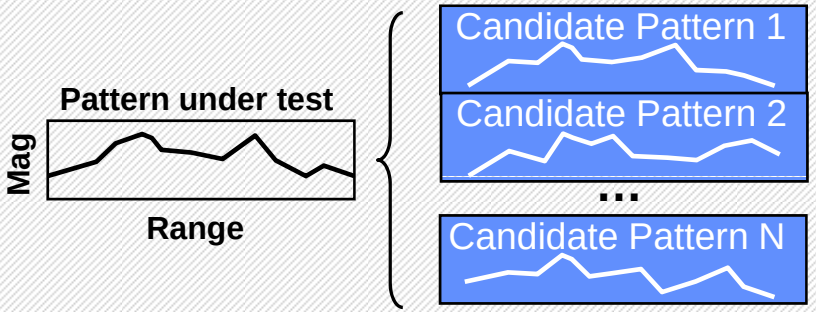
Genetic Algorithm



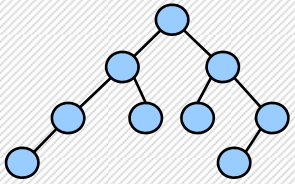
- Evaluate each chromosome
- Select chromosomes for next generation
- Crossover: randomly pair up chromosomes and exchange portions
- Mutation: randomly change each chromosome

Pattern Match

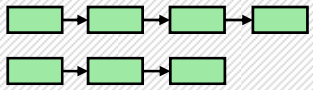
- Compute best match for a pattern out of set of candidate patterns
  - Uses weighted mean-square error



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

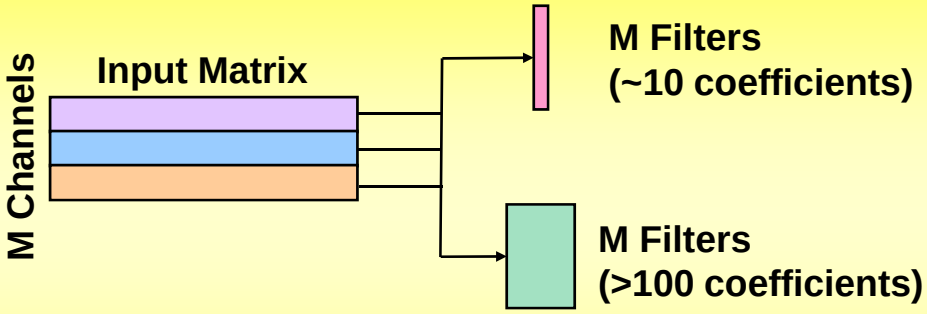
* Numbers denote memory content

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

HPEC Challenge

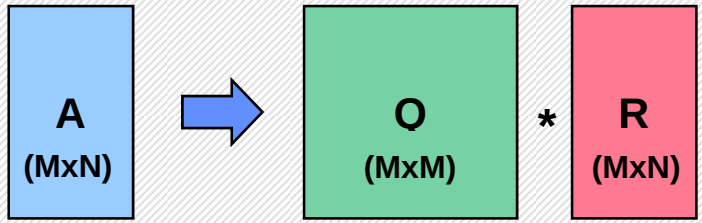
Signal and Image Processing Kernels

FIR



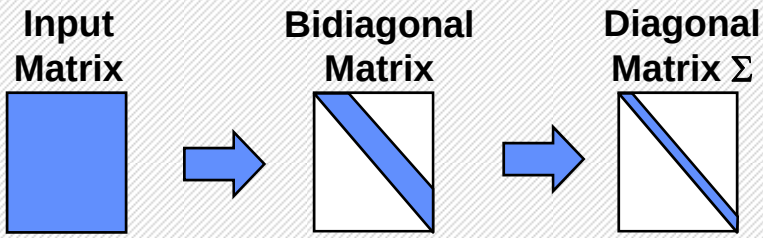
- Bank of filters applied to input data
- FIR filters implemented in time and frequency domain

QR



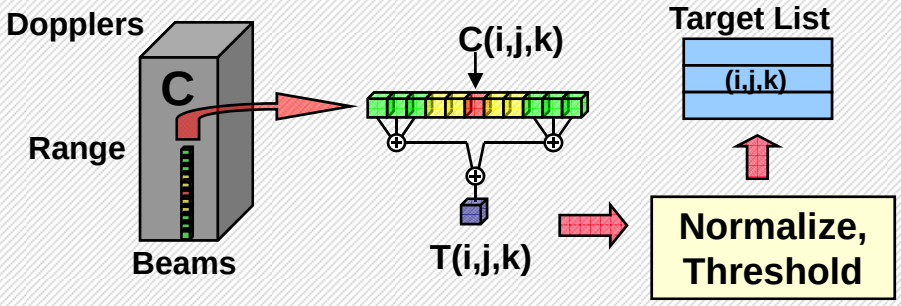
- Computes the factorization of an input matrix, A=QR
- Implementation uses Fast Givens algorithm

SVD



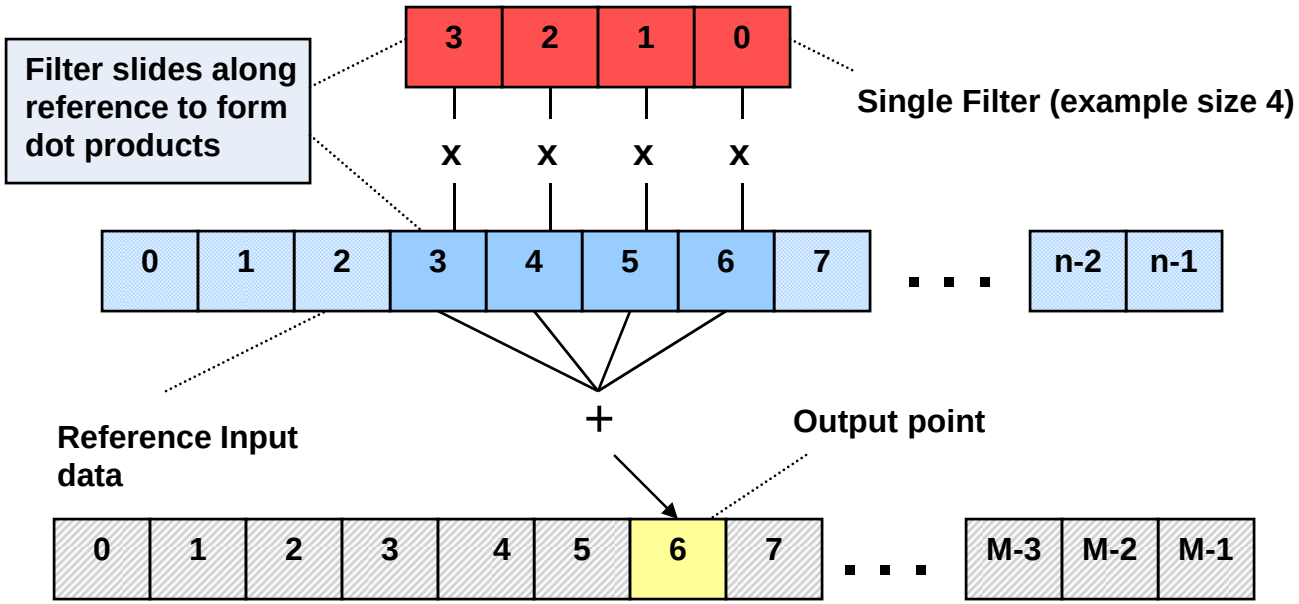
- Produces decomposition of an input matrix, $X=U\Sigma V^H$
- Classic Golub-Kahan SVD implementation

CFAR



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

Time Domain FIR Algorithm



- Number of Operations:
  - k – Filter size
  - n – Input size
  - nf - number of filters
  - Total FOPs: ~ 8 x nf x n x k
- Output Size: n + k - 1

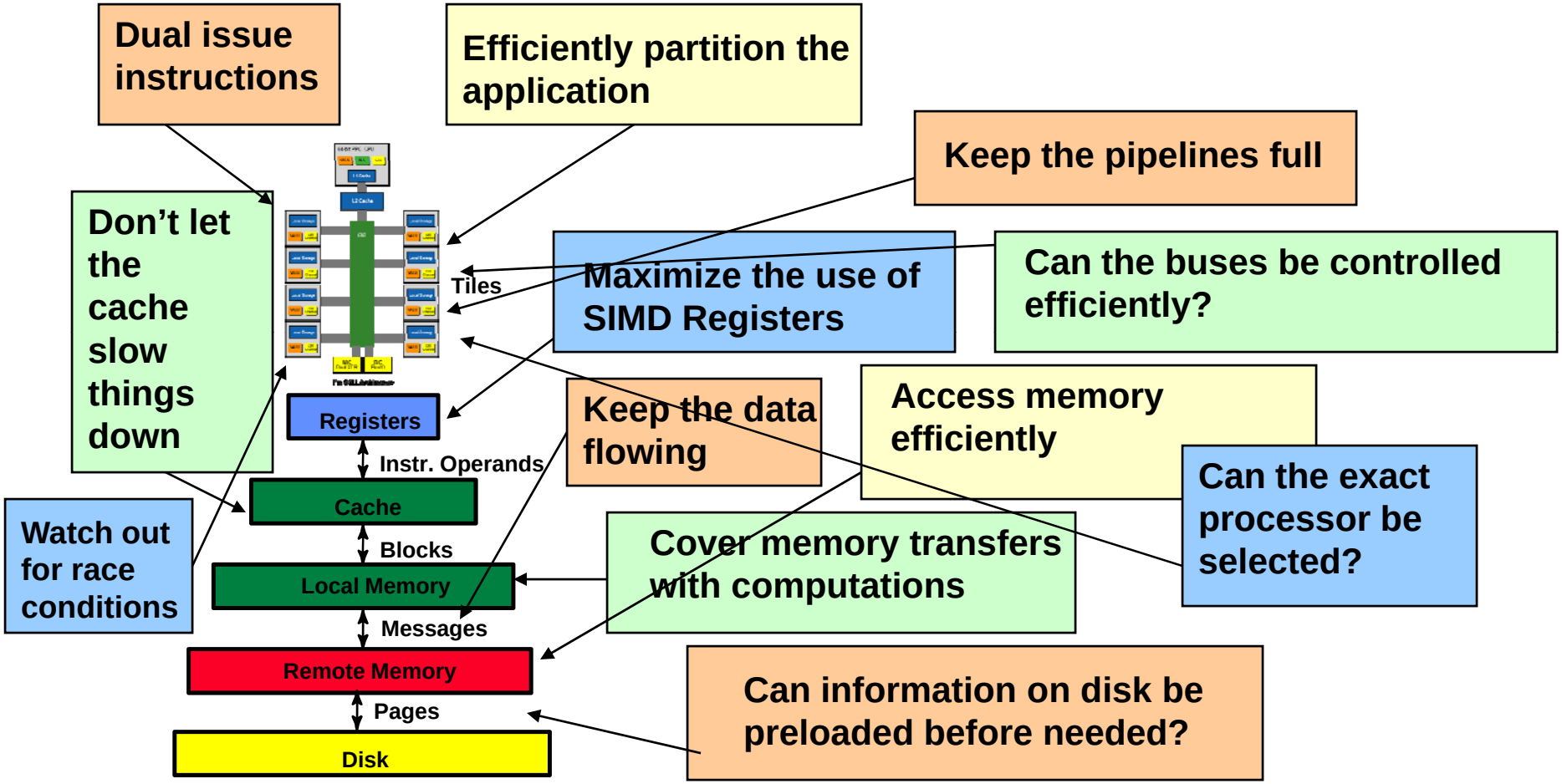
- TDFIR uses complex data
- TDFIR uses a bank of filters
  - Each filter is used in a tapered convolution
  - A convolution is a series of dot products

HPEC Challenge Parameters TDFIR

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

•FIR is one of the best ways to demonstrate FLOPS

Performance Challenges



- Price of performance is increased programming complexity

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

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

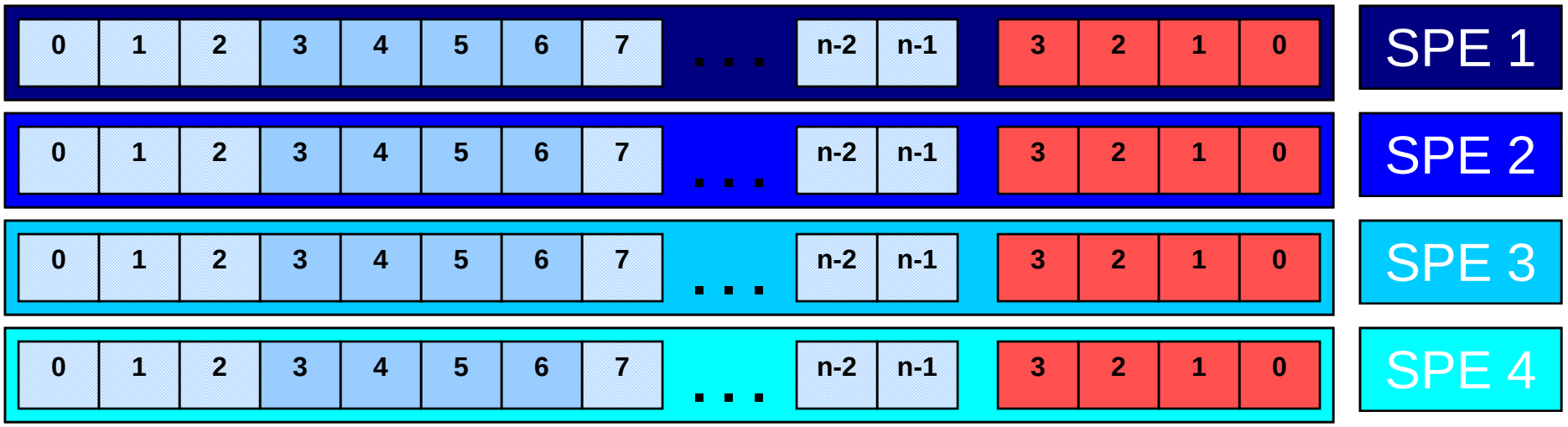
```
/* Fourth kernel point contribution  */
fma $50,$30,$66,$50          /* reout''' + rein * rek [4i+16j+3]-[4i+16j+6]   */
lqd $45,32($15)              /* load imin[16j+24]-[16j+27]                    */
fma $51,$31,$66,$51          /* imout''' + rein * imk [4i+16j+3]-[4i+16j+6]   */
shufb $66,$40,$42,$16        /* 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+31]              */
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

- **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



- **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

- Overview
- **HPEC Challenge Benchmarks**
  - *Time-Domain FIR Filter*
  - ***Results***
- Parallel Vector Tile Optimizing Library
- Summary

Mercury Cell Processor Test System

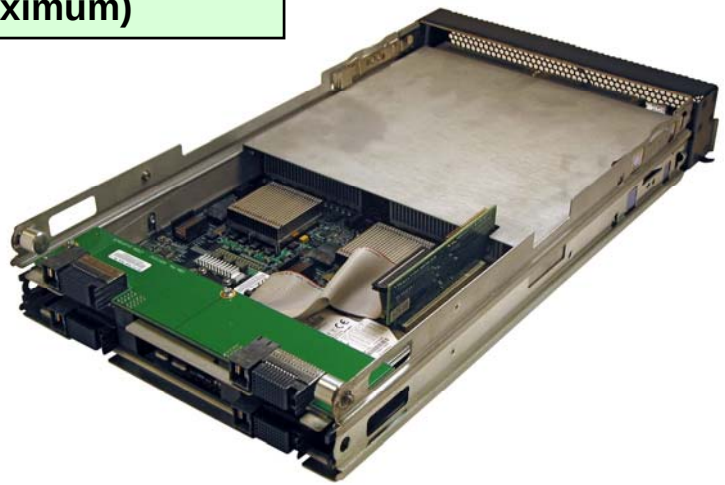
Mercury Cell Processor System

- 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

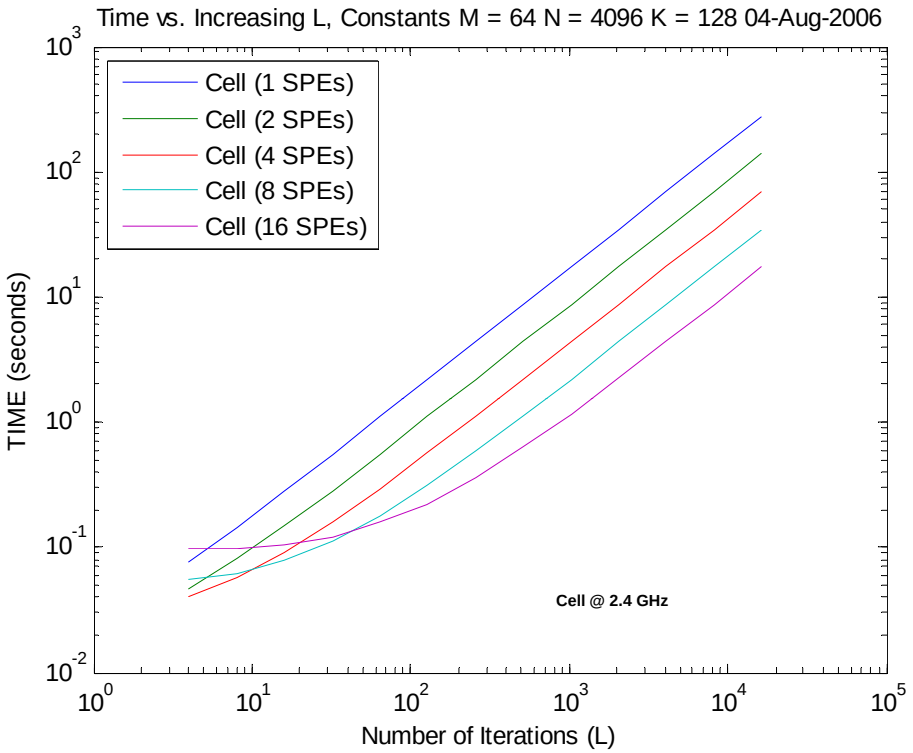
•Each Cell has 153.6 GFLOPS (single precision)
– 307.2 for system @ 2.4 GHz (maximum)

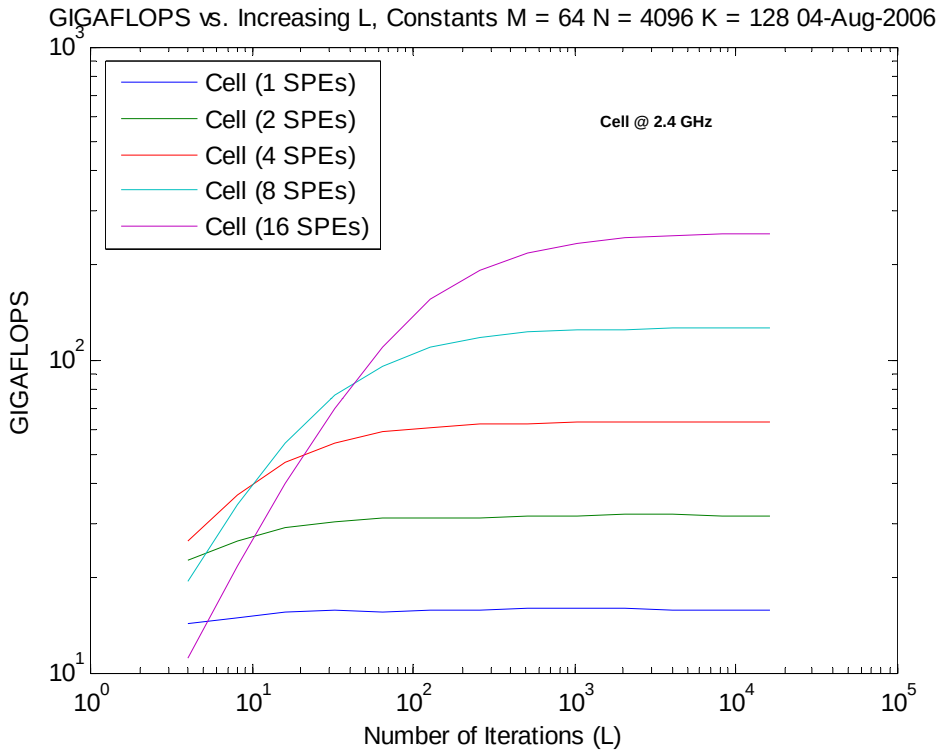
Software includes:

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



Performance Time Domain FIR





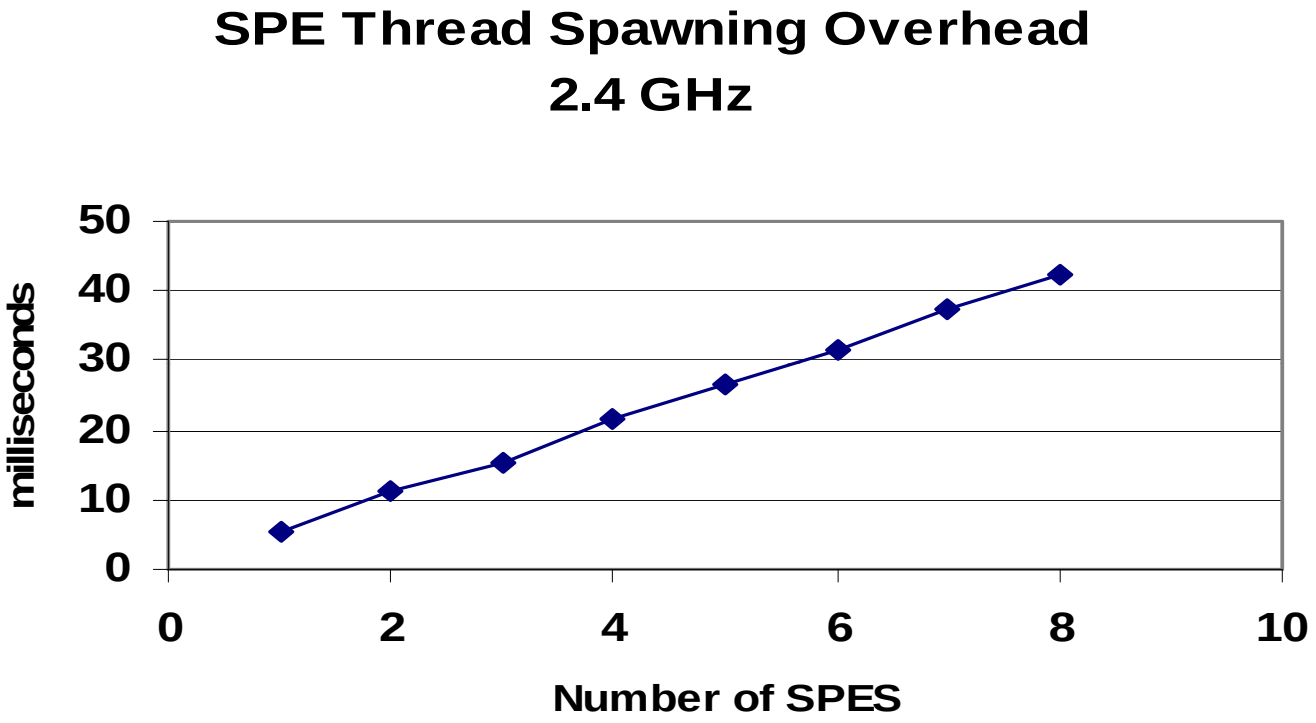
Set 1 has a bank of 64 size 128 filters with size 4096 input vectors

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

Maximum GFLOPS for TDFIR #1

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

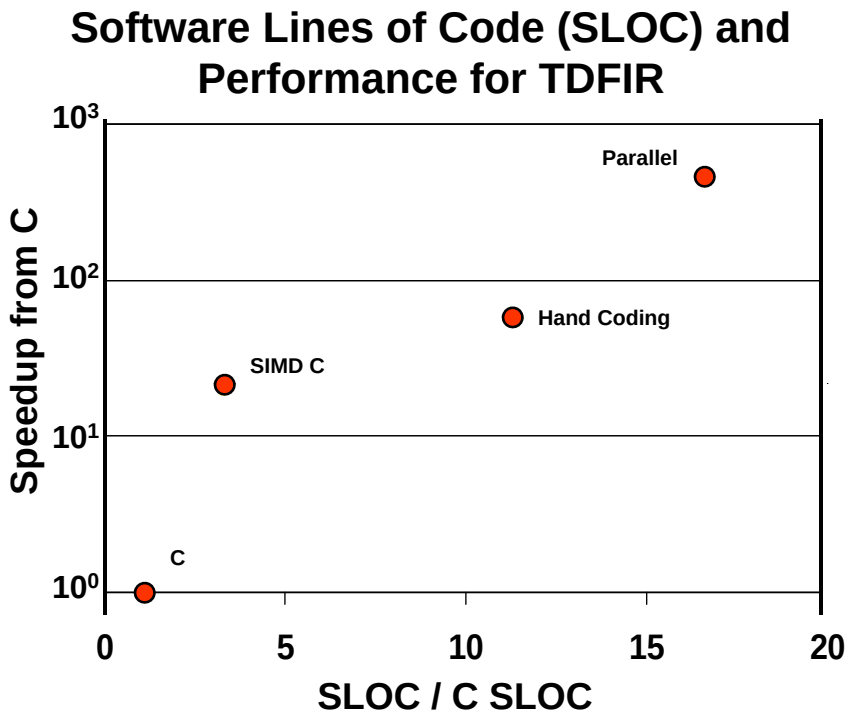
Overhead



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

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



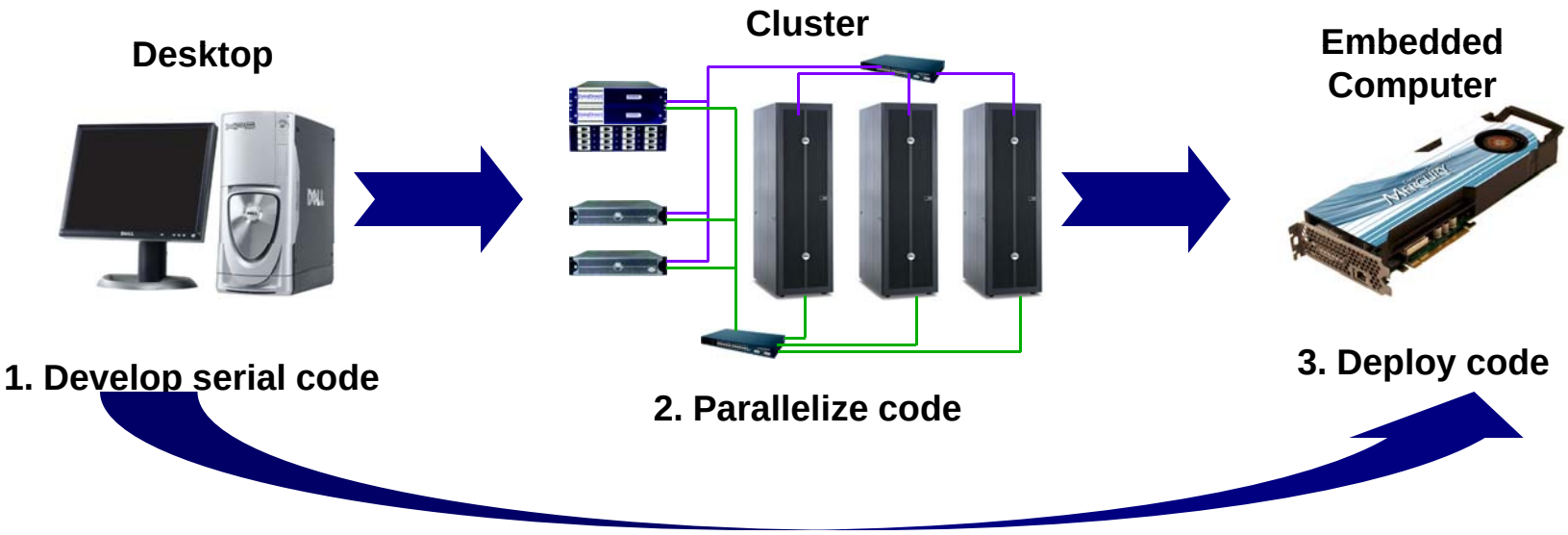
- **Clear tradeoff between performance and effort**
  - C code simple, poor performance
  - SIMD C, more complex to code, reasonable performance
  - Hand coding, very complex, excellent performance

Outline

- Overview
- HPEC Challenge Benchmarks
- **Parallel Vector Tile Optimizing Library (PVTOL)**
  - *PVTOL Architecture*
  - *PVTOL Development Cycle*
  - *Contributing Technologies*
- Summary

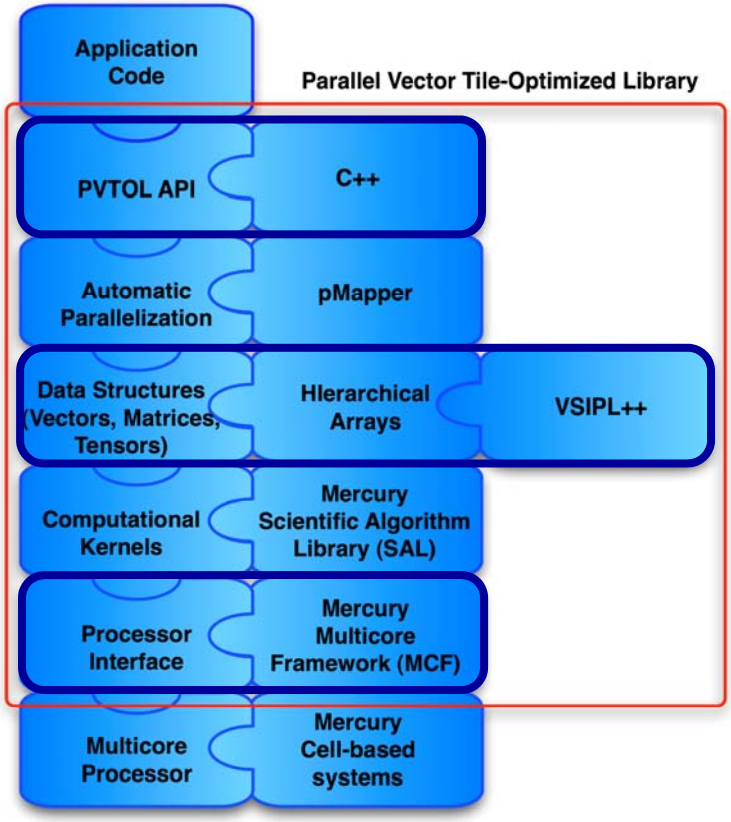
Parallel Vector Tile Optimizing Library (PVTOL)

- PVTOL is a portable and scalable middleware library for multicore processors
- Enables incremental development



Make parallel programming as easy as serial programming

PVTOL Architecture



- **Performance**
  - Achieves high performance
- **Portability**
  - Built on standards, e.g. VSIPL++
- **Productivity**
  - Minimizes effort at user level

PVTOL preserves the simple load-store programming model in software

PVTOL Development Process

Serial PVTOL code

```
void main(int argc, char *argv[]) {
 // Initialize PVTOL
 process pvtol(argc, argv);




 // Create input, weights, and output matrices
 typedef Dense<2, float, tuple<0, 1> > dense_block_t;
 typedef Matrix<float, dense_block_t, LocalMap> matrix_t;
 matrix_t input(num_vects, len_vect),
          filts(num_vects, len_vect),
          output(num_vects, len_vect);

 // Initialize arrays
 ...

 // Perform TDFIR filter
 output = tdfir(input, filts);
}
```

PVTOL Development Process

Parallel PVTOL code

```
void main(int argc, char *argv[]) {
 // Initialize PVTOL
 process pvtol(argc, argv);




 // Add parallel map
 RunTimeMap map1(...);

 // Create input, weights, and output matrices
 typedef Dense<2, float, tuple<0, 1> > dense_block_t;
 typedef Matrix<float, dense_block_t, RunTimeMap> matrix_t;
 matrix_t input(num_vects, len_vect, map1),
          filts(num_vects, len_vect, map1),
          output(num_vects, len_vect, map1);

 // Initialize arrays
 ...

 // Perform TDFIR filter
 output = tdfir(input, filts);
}
```

PVTOL Development Process

Embedded PVTOL code

```
void main(int argc, char *argv[]) {
 // Initialize PVTOL
 process pvtol(argc, argv);

 // Add hierarchical map
 RunTimeMap map2(...);

 // Add parallel map
 RunTimeMap map1(..., map2);

 // Create input, weights, and output matrices
 typedef Dense<2, float, tuple<0, 1> > dense_block_t;
 typedef Matrix<float, dense_block_t, RunTimeMap> matrix_t;
 matrix_t input(num_vects, len_vect, map1),
          filts(num_vects, len_vect, map1),
          output(num_vects, len_vect, map1);

 // Initialize arrays
 ...

 // Perform TDFIR filter
 output = tdfir(input, filts);
}
```

PVTOL Development Process

Automapped PVTOL code

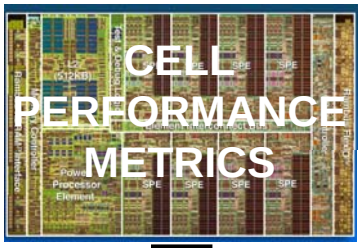
```
void main(int argc, char *argv[]) {
 // Initialize PVTOL
 process pvtol(argc, argv);




 // Create input, weights, and output matrices
 typedef Dense<2, float, tuple<0, 1> > dense_block_t;
 typedef Matrix<float, dense_block_t, AutoMap> matrix_t;
 matrix_t input(num_vects, len_vect),
          filts(num_vects, len_vect),
          output(num_vects, len_vect);

 // Initialize arrays
 ...

 // Perform TDFIR filter
 output = tdfir(input, filts);
}
```

Outline

- Overview
- HPEC Challenge Benchmarks
- **Parallel Vector Tile Optimizing Library (PVTOL)**
  - *PVTOL Architecture*
  - *PVTOL Development Cycle*
  - ***Contributing Technologies***
- Summary

pMapper Automated Mapping System**
Cell Processor Simulation

- Simulate the Cell processor using pMapper simulator infrastructure
- Use pMapper to automate mapping and predict performance on the Cell



PERFORM. MODEL

PROGRAM SPEC

PERFORM. MODEL

EXPERT MAPPING SYSTEM

ATLAS

SIGNAL FLOW EXTRACTOR

SIGNAL FLOW GRAPH

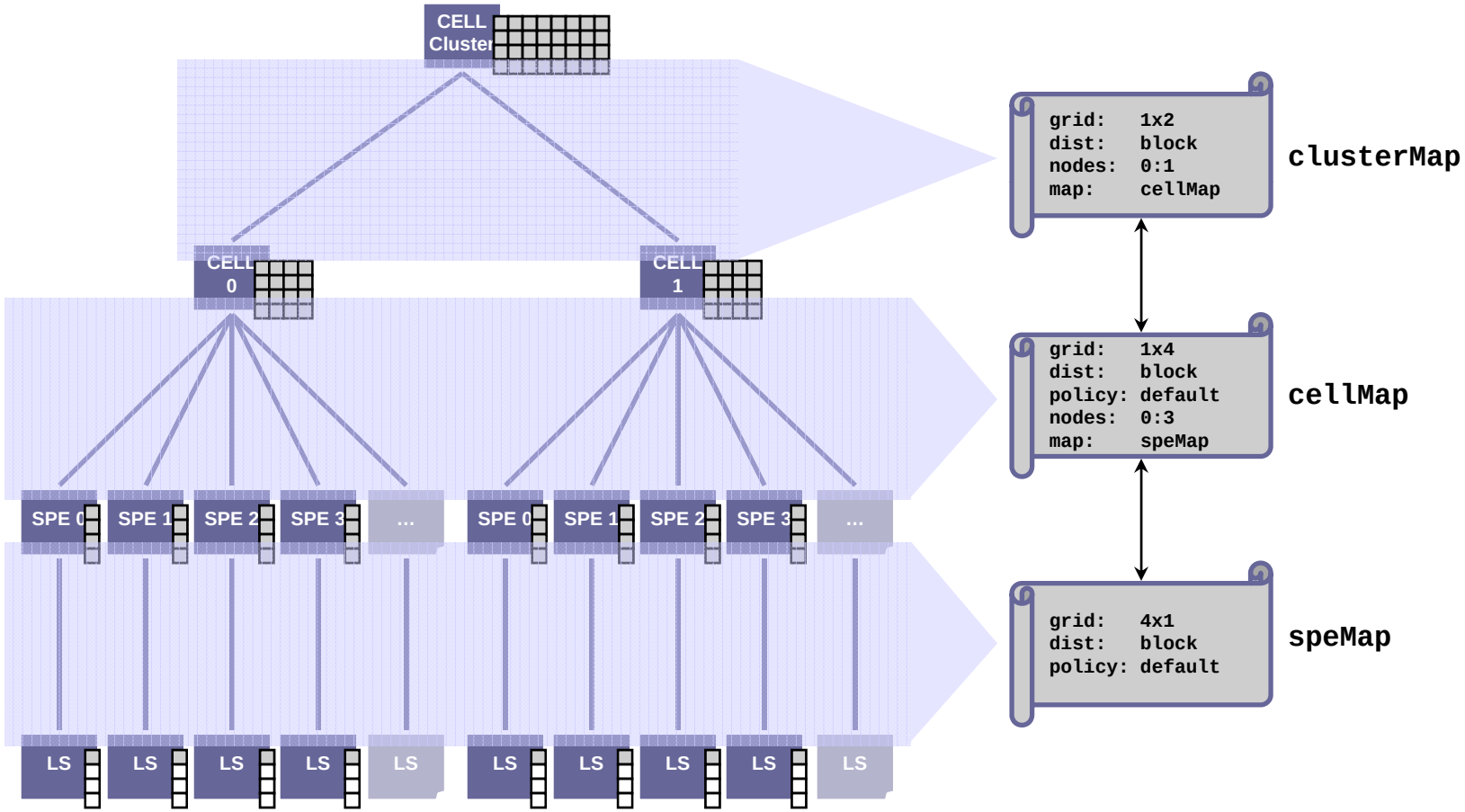
CELL SIMULATOR

PROGRAM TIMING

** Note: Patent process underway

Hierarchical Arrays

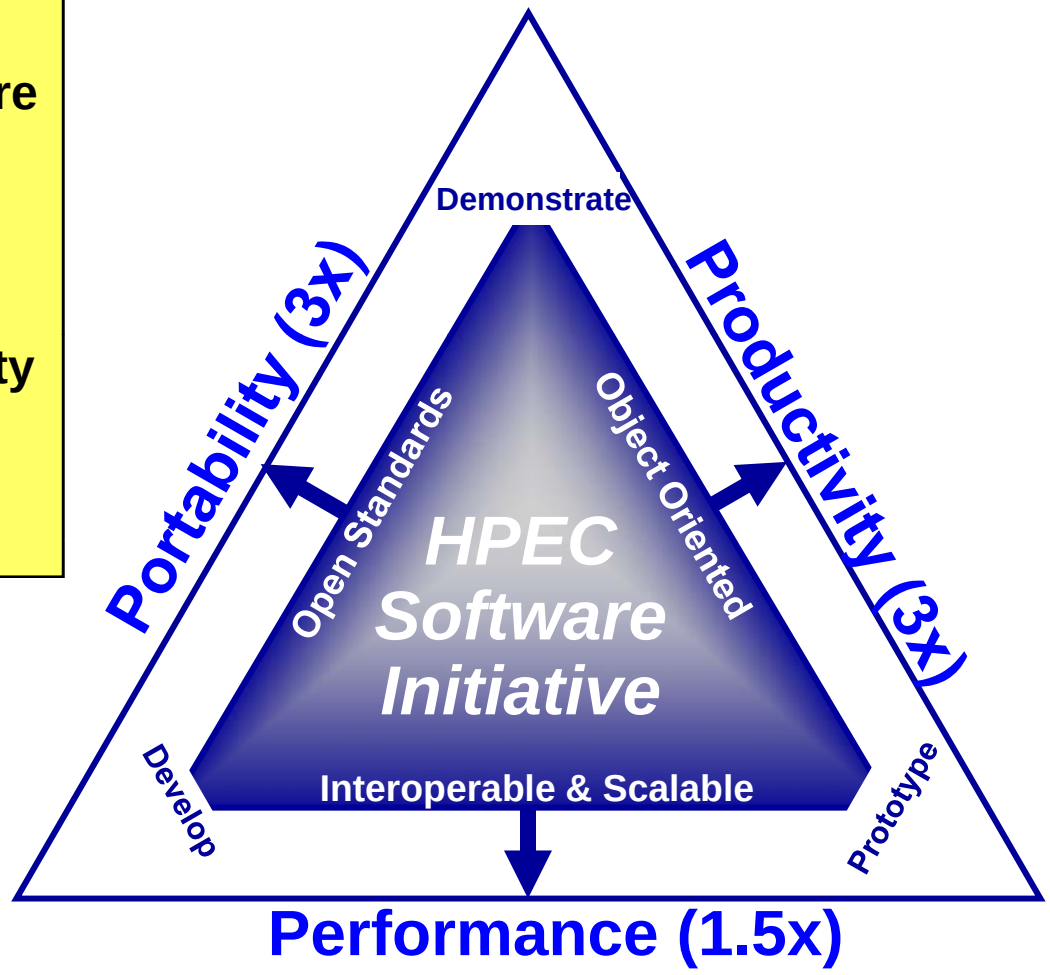
- Hierarchical arrays hide details of the processor and memory hierarchy
- Hierarchical maps concisely describe data distribution at each level



The High Performance Embedded Computing Software Initiative

Program Goals

- Develop and integrate software technologies for embedded parallel systems to address portability, productivity, and performance
- Engage acquisition community to promote technology insertion
- Deliver quantifiable benefits

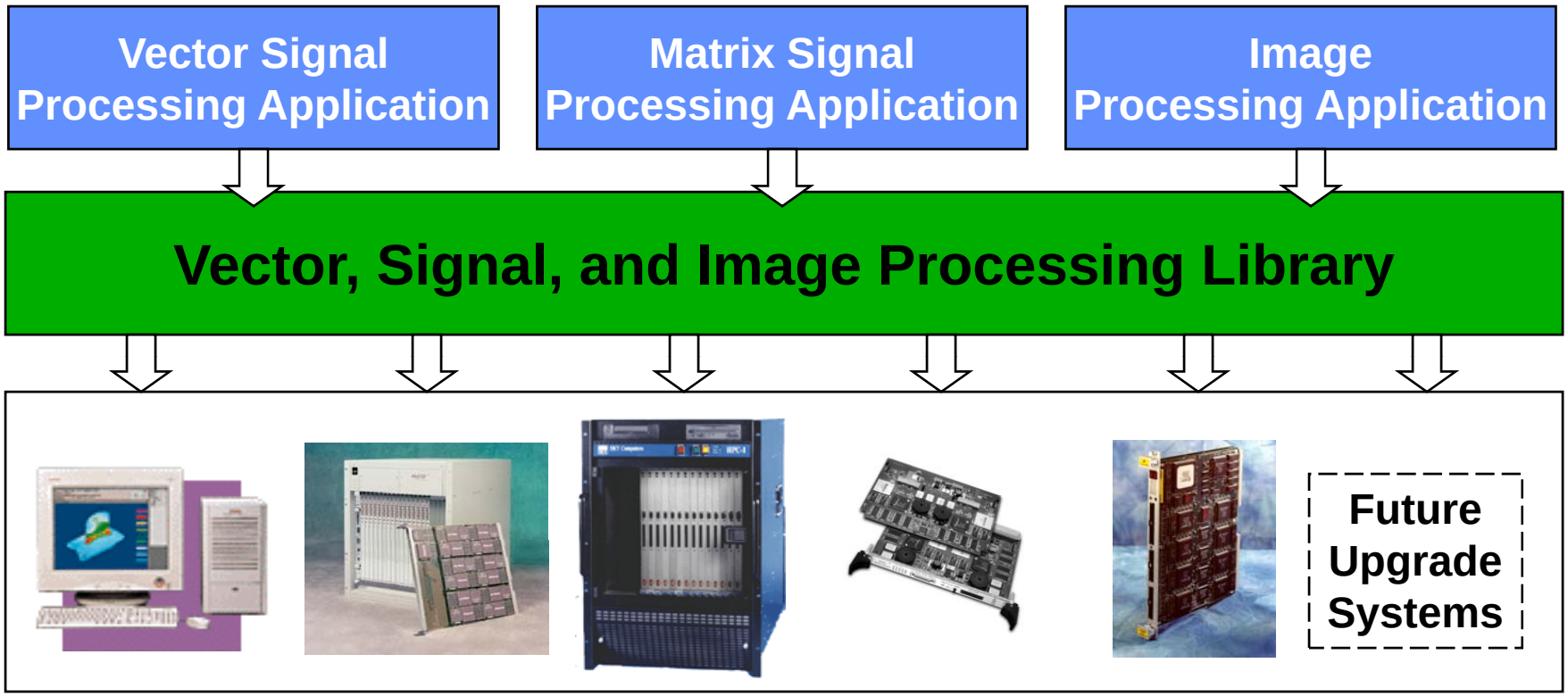


Portability: reduction in lines-of-code to change port/scale to new system

Productivity: reduction in overall lines-of-code

Performance: computation and communication benchmarks

VSIPL++



- Portable to workstations, embedded systems, FPGAs with minimal performance cost
- Applicable to simple and complex applications
- Easier upgrade cycle
- Reduced development time and cost

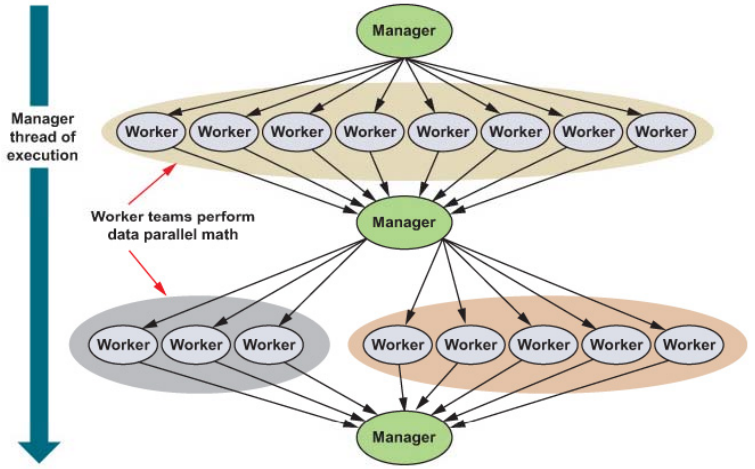
- Parallelism built in
- Object Oriented
- Basic Arithmetic, Matrix Algebra, Signal Processing, and Equation Solvers

VSIPL++ is freely available

www.hpec-si.org

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

- **Leveraging vendor libraries reduces development time**
  - Provides optimization
  - Less debugging of application

Summary

- With basic tools high performance is achievable from the Cell processor
  - Hard to program: SIMD C extensions or assembly code are required
  - Development and debugging time can be long

- PVTOL is making programming easier for Cell
  - Leverages existing technologies
  - Packages common kernels needed by users
  - API simplifies application code

Backup