PVTOL:

Designing Portability, Productivity and Performance for Multicore Architectures

Hahn Kim, Nadya Bliss, Jim Daly, Karen Eng, Jeremiah Gale, James Geraci, Ryan Haney, Jeremy Kepner, Sanjeev Mohindra, Sharon Sacco, Eddie Rutledge

HPEC 2008

25 September 2008

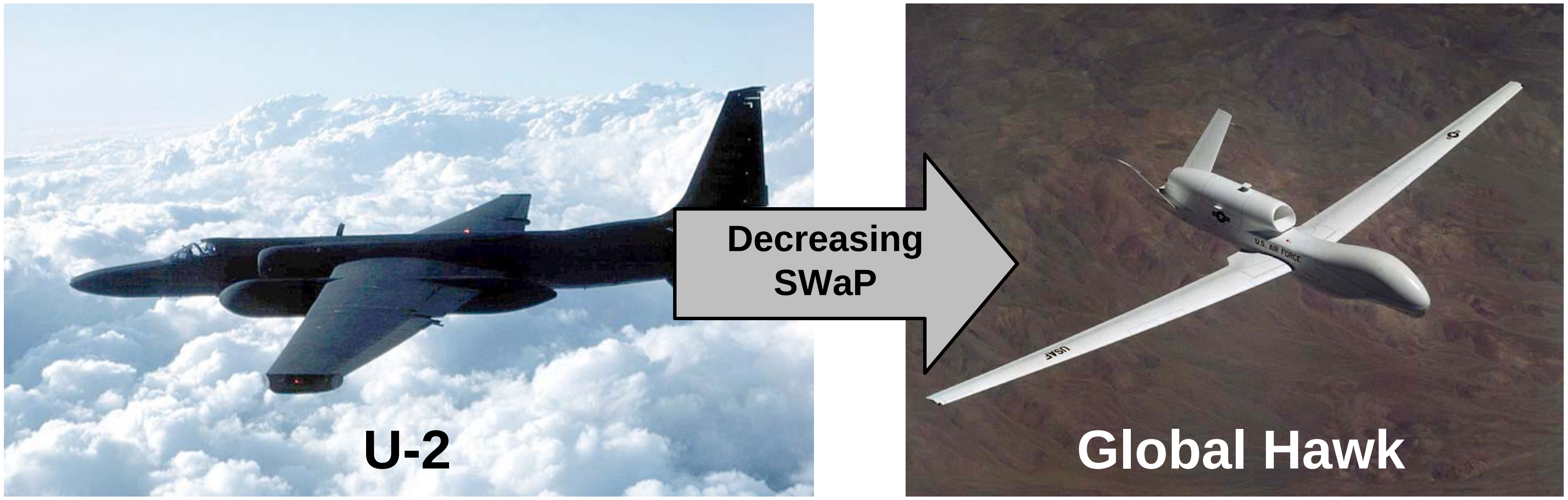
This work is sponsored by the Department of the Air Force under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendataions are those of the author and are not necessarily endorsed by the United States Government.

Outline

- **Background**
  - **Motivation**
  - **Multicore Processors**
  - **Programming Challenges**
- **Tasks & Conduits**
- **Maps & Arrays**
- **Results**
- **Summary**

SWaP* for Real-Time Embedded Systems

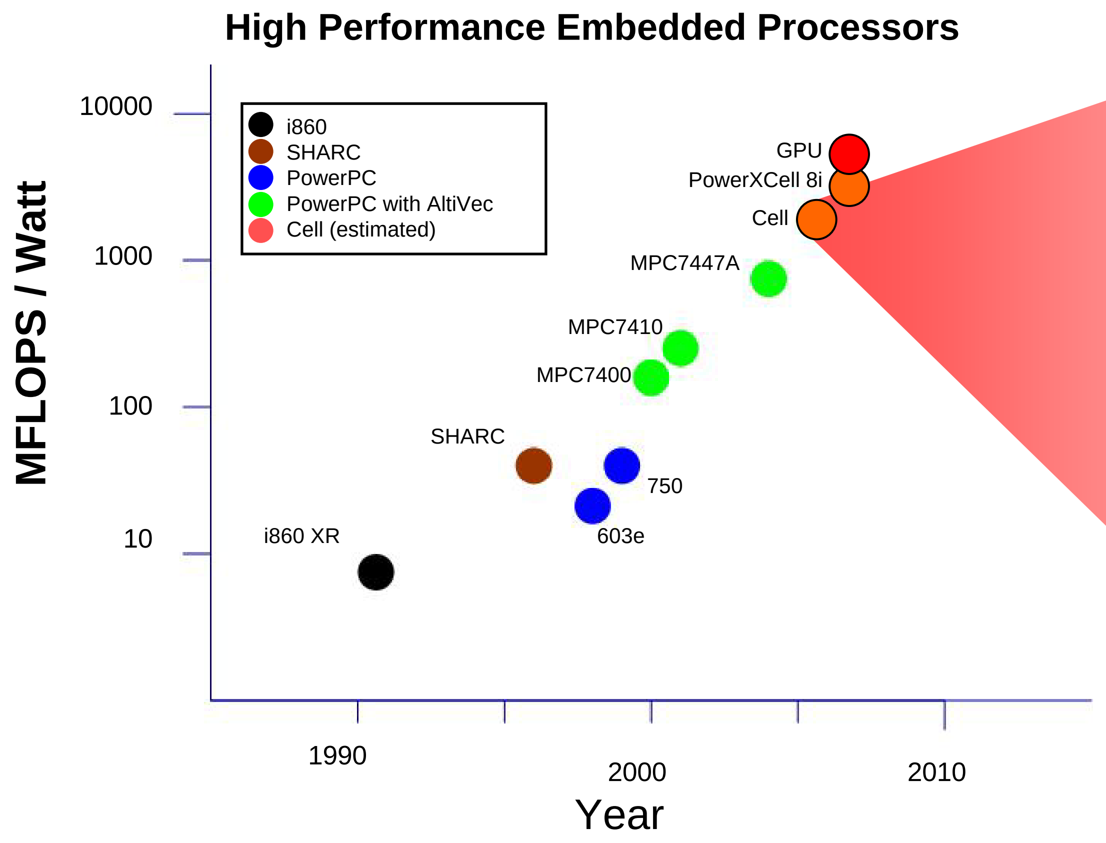
- Modern DoD sensors continue to increase in fidelity and sampling rates
- Real-time processing will always be a requirement

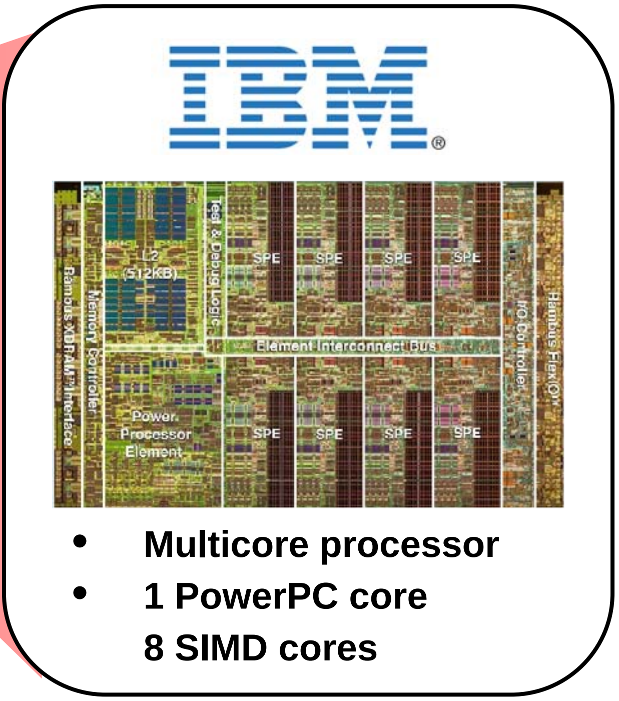


Modern sensor platforms impose tight SWaP requirements on real-time embedded systems

* SWaP = Size, Weight and Power

Embedded Processor Evolution





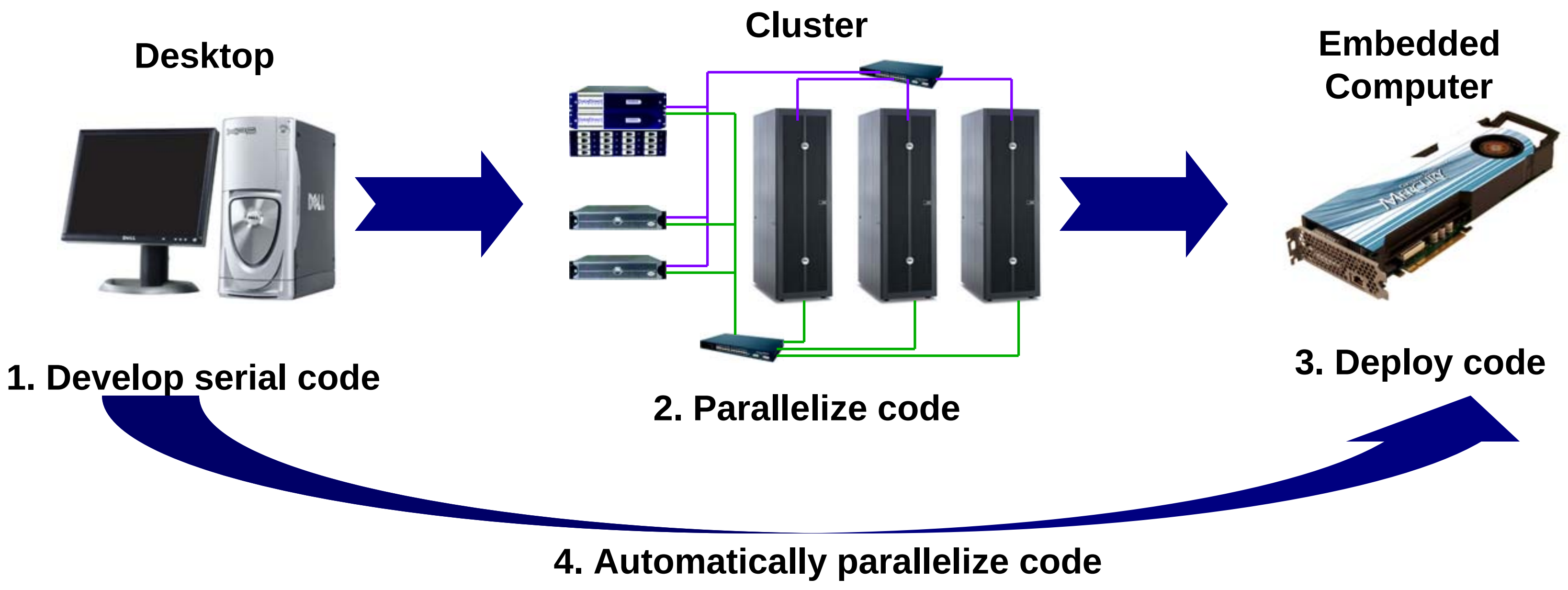
- Multicore processor
- 1 PowerPC core
  8 SIMD cores

- 20 years of exponential growth in FLOPS / W
- Must switch architectures every ~5 years
- Current high performance architectures are multicore

Multicore processors help achieve performance requirements within tight SWaP constraints

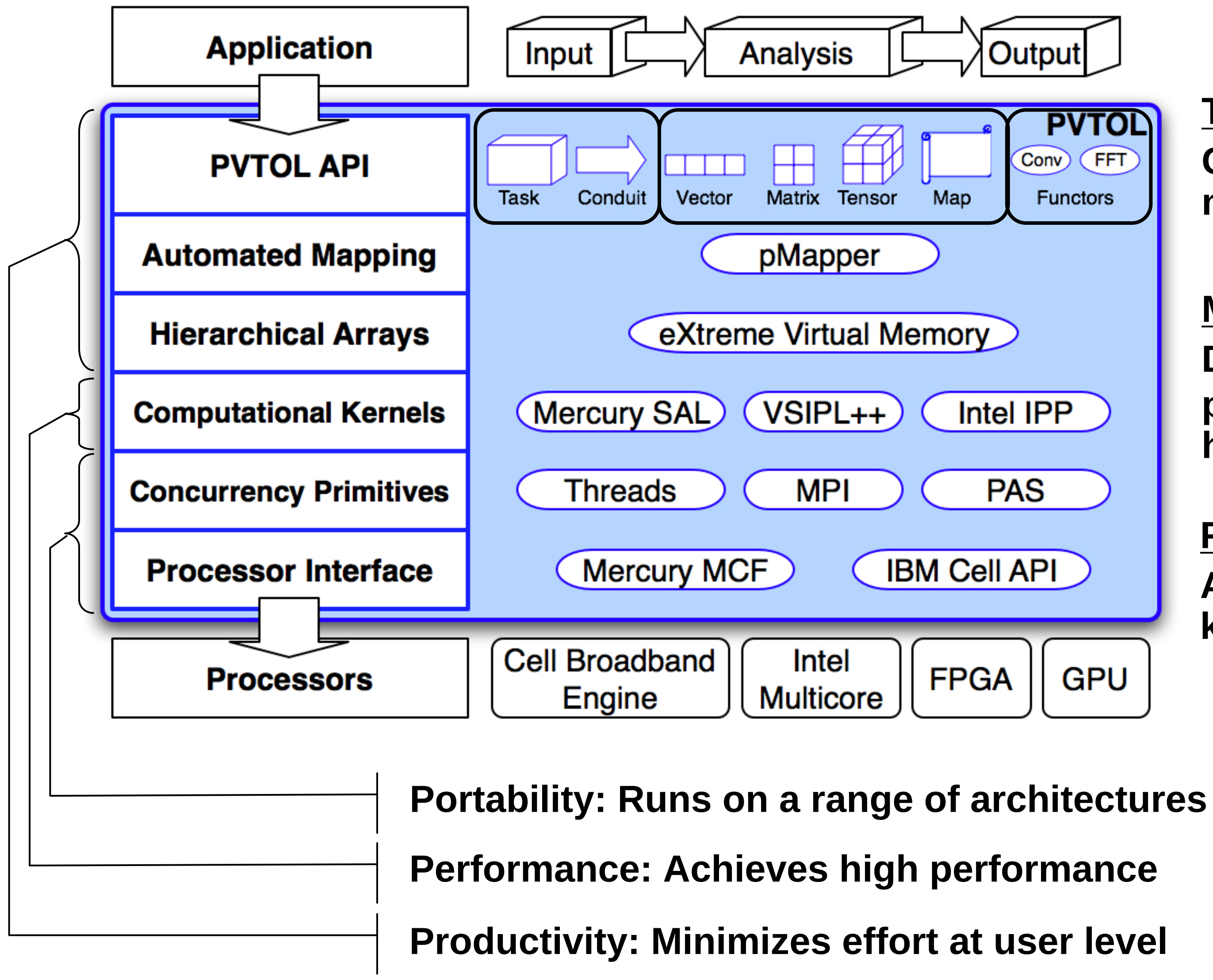
Parallel Vector Tile Optimizing Library

- **PVTOL is a portable and scalable middleware library for multicore processors**
- **Enables unique software development process for real-time signal processing applications**



Make parallel programming as easy as serial programming

PVTOL Architecture



Portability: Runs on a range of architectures

Performance: Achieves high performance

Productivity: Minimizes effort at user level

<u>Tasks & Conduits</u>

Concurrency and data movement

<u>Maps & Arrays</u>

Distribute data across processor and memory hierarchies

<u>Functors</u>

Abstract computational kernels into objects

Outline

- Background
- Tasks & Conduits
- Maps & Arrays
- Results
- Summary

Multicore Programming Challenges

Inside the Box

Desktop

Embedded Board

- Threads
  - Pthreads
  - OpenMP
- Shared memory
  - Pointer passing
  - Mutexes, condition variables

Outside the Box

Cluster

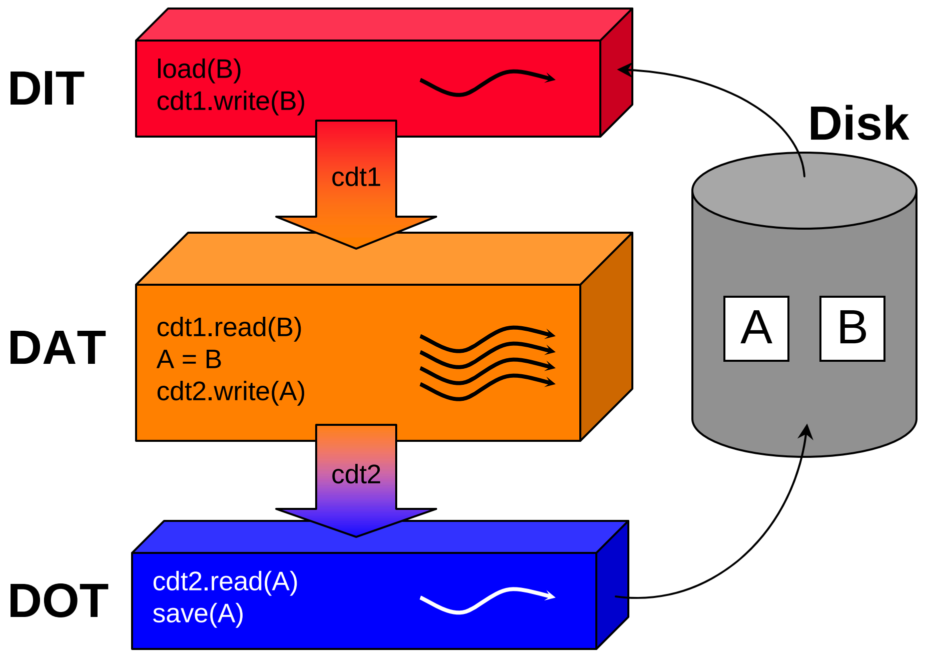
Embedded Multicomputer

- Processes
  - MPI (MPICH, Open MPI, etc.)
  - Mercury PAS
- Distributed memory
  - Message passing

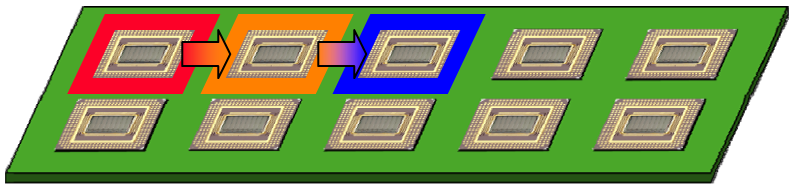
PVTOL provides consistent semantics for both multicore and cluster computing

Tasks & Conduits

- Tasks provide concurrency
  - Collection of 1+ threads in 1+ processes
  - Tasks are SPMD, i.e. each thread runs task code
- Task Maps specify locations of Tasks
- Conduits move data
  - Safely move data
  - Multibuffering
  - Synchronization



DIT	Read data from source (1 thread)
DAT	Process data (4 threads)
DOT	Output results (1 thread)
Conduits	Connect DIT to DAT and DAT to DOT



MIT Lincoln Laboratory

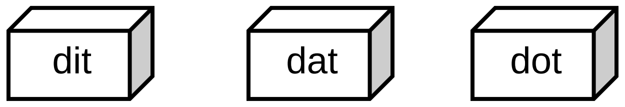
* DIT – Data Input Task, DAT – Data Analysis Task, DOT – Data Output Task

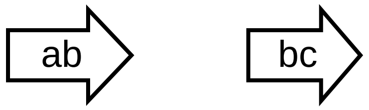
Pipeline Example

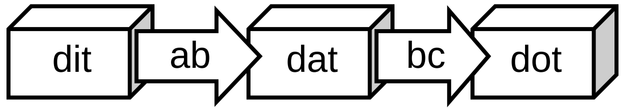
DIT-DAT-DOT

```
int main(int argc, char** argv)
{
   // Create maps (omitted for brevity)
   ...

   // Create the tasks
   Task<Dit> dit("Data Input Task", ditMap);
   Task<Dat> dat("Data Analysis Task", datMap);
   Task<Dot> dot("Data Output Task", dotMap);

   // Create the conduits
   Conduit<Matrix <double> > ab("A to B Conduit");
   Conduit<Matrix <double> > bc("B to C Conduit");

   // Make the connections
   dit.init(ab.getWriter());
   dat.init(ab.getReader(), bc.getWriter());
   dot.init(bc.getReader());

   // Complete the connections
   ab.setupComplete(); bc.setupComplete();

   // Launch the tasks
   dit.run(); dat.run(); dot.run();

   // Wait for tasks to complete
   dit.waitTillDone();
   dat.waitTillDone();
   dot.waitTillDone();
}
```

Main function creates tasks, connects tasks with conduits and launches the task computation







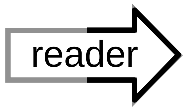
Pipeline Example

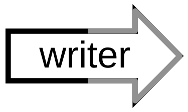
Data Analysis Task (DAT)

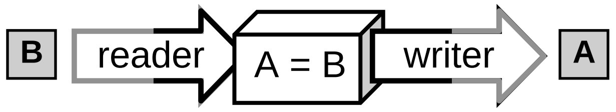
```
class Dat
{
private:
   Conduit<Matrix <double> >::Reader m_Reader;
   Conduit<Matrix <double> >::Writer m_Writer;

public:
   void init(Conduit<Matrix <double> >::Reader& reader,
             Conduit<Matrix <double> >::Writer& writer)
   {
      // Get data reader for the conduit
      reader.setup(tr1::Array<int, 2>(ROWS, COLS));
      m_Reader = reader;

      // Get data writer for the conduit
      writer.setup(tr1::Array<int, 2>(ROWS, COLS));
      m_Writer = writer;
   }

   void run()
   {
      Matrix <double>& B = m_Reader.getData();
      Matrix <double>& A = m_Writer.getData();
      A = B;
      m_reader.releaseData();
      m_writer.releaseData();
   }
};
```

Tasks read and write data using Reader and Writer interfaces to Conduits

Readers and Writer provide handles to data buffers







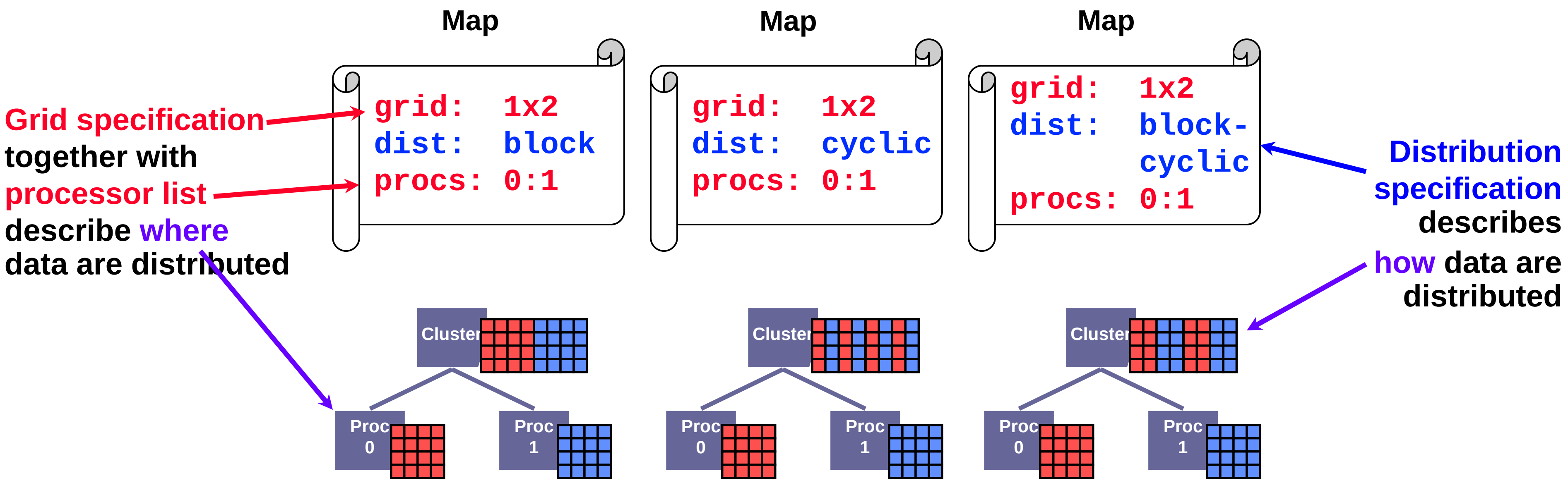
Outline

- Background
- Tasks & Conduits
- **Maps & Arrays**
  - Hierarchy
  - Functors
- Results
- Summary

Map-Based Programming

- A map is an assignment of blocks of data to processing elements
- Maps have been demonstrated in several technologies

Technology	Organization	Language	Year
Parallel Vector Library	MIT-LL*	C++	2000
pMatlab	MIT-LL	MATLAB	2003
VSIPL++	HPEC-SI†	C++	2006



Map
grid: 1x2
dist: block
procs: 0:1
Map
grid: 1x2
dist: cyclic
procs: 0:1
Map
grid: 1x2
dist: block-cyclic
procs: 0:1
Grid specification together with processor list describe where data are distributed
Distribution specification describes how data are distributed
Cluster
Proc 0
Proc 1

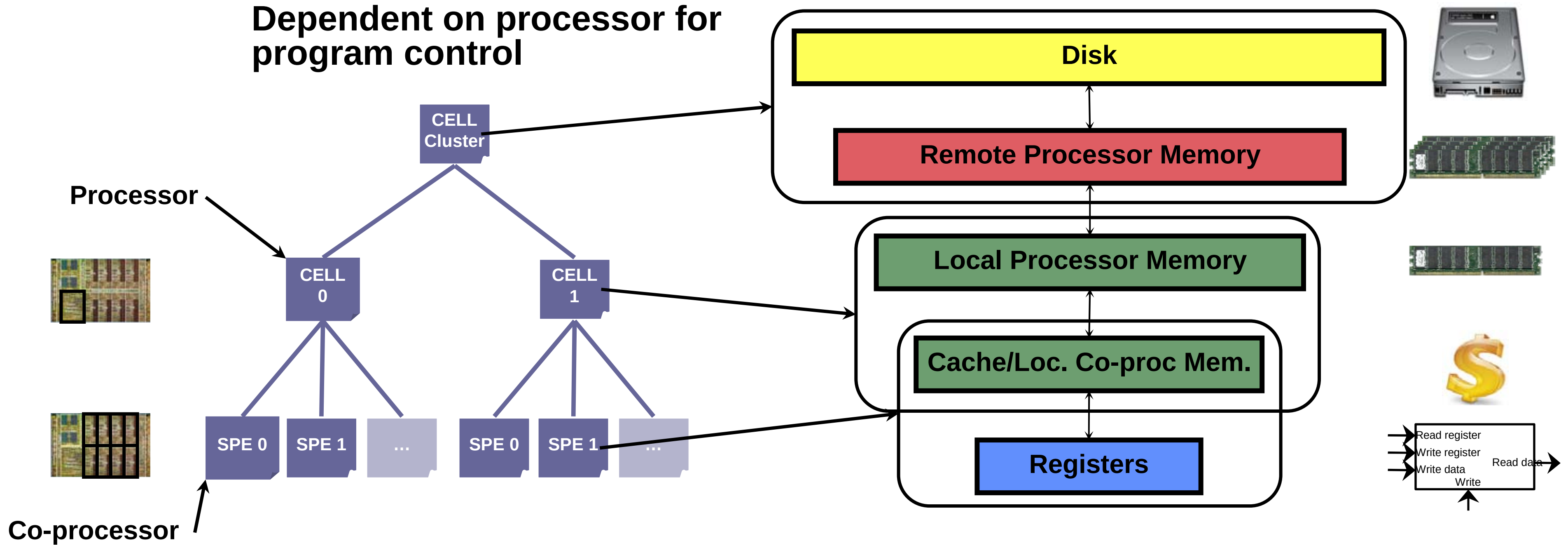
* MIT Lincoln Laboratory † High Performance Embedded Computing Software Initiative

PVTOL Machine Model

- Processor Hierarchy
  - Processor:

    Scheduled by OS
  - Co-processor:

    Dependent on processor for program control
- Memory Hierarchy
  - Each level in the processor hierarchy can have its own memory

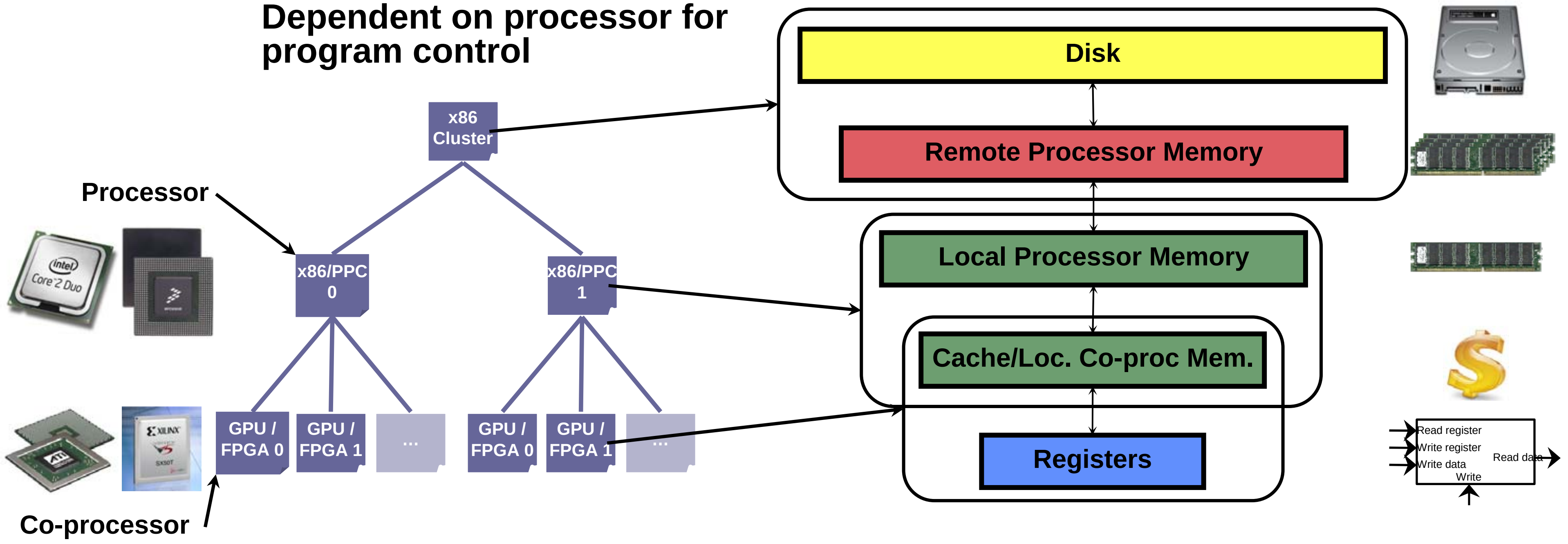


PVTOL extends maps to support hierarchy

PVTOL Machine Model

- **Processor Hierarchy**
  - **Processor:**
    Scheduled by OS
  - **Co-processor:**
    Dependent on processor for program control

- **Memory Hierarchy**
  - **Each level in the processor hierarchy can have its own memory**



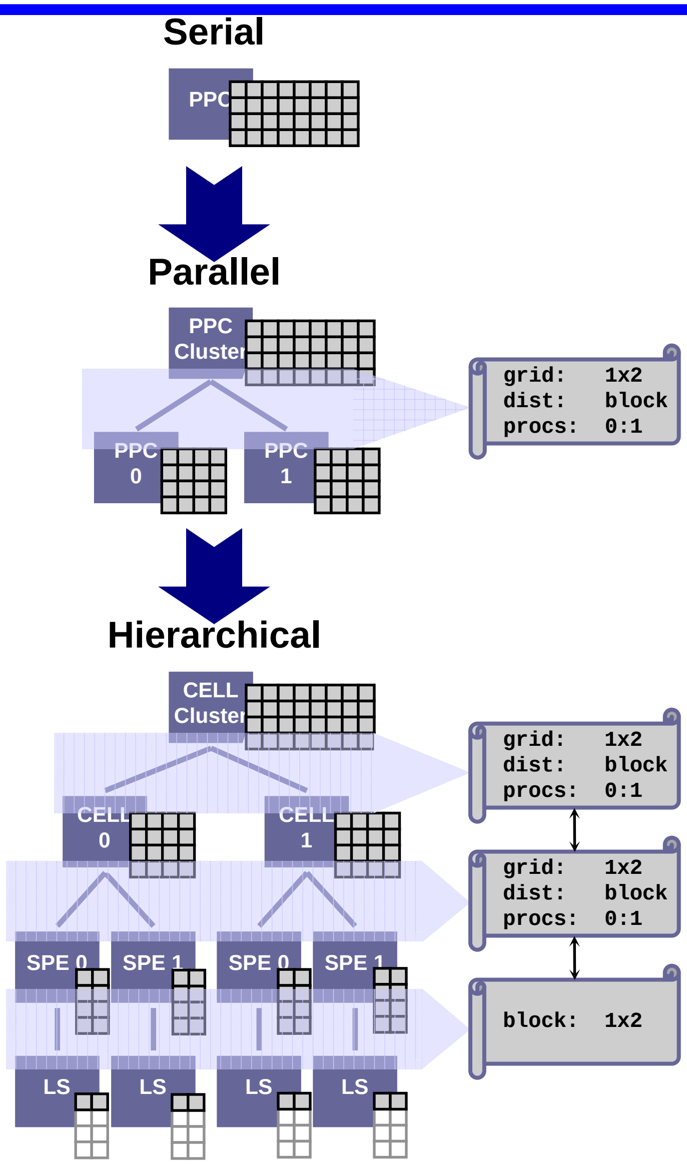
Semantics are the same across different architectures

Hierarchical Maps and Arrays

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

Program Flow

1. Define a Block
   - Data type, index layout (e.g. row-major)
2. Define a Map for each level in the hierarchy
   - Grid, data distribution, processor list
3. Define an Array for the Block
4. Parallelize the Array with the Hierarchical Map (optional)
5. Process the Array

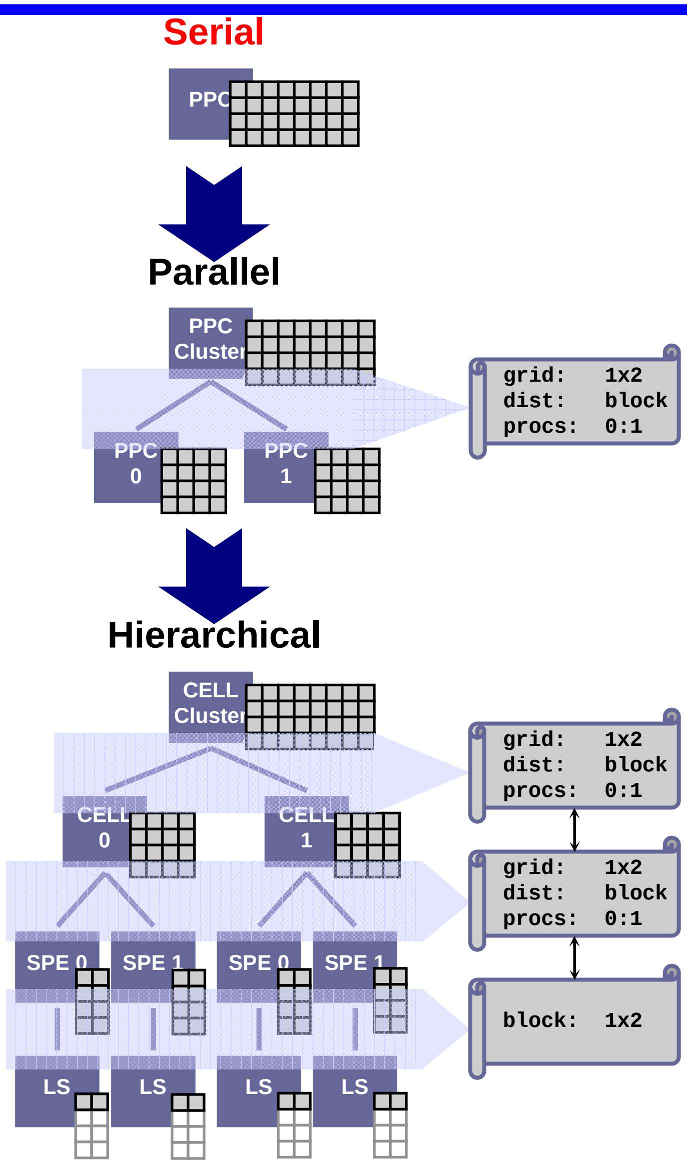


MIT Lincoln Laboratory

Hierarchical Maps and Arrays

Example - Serial

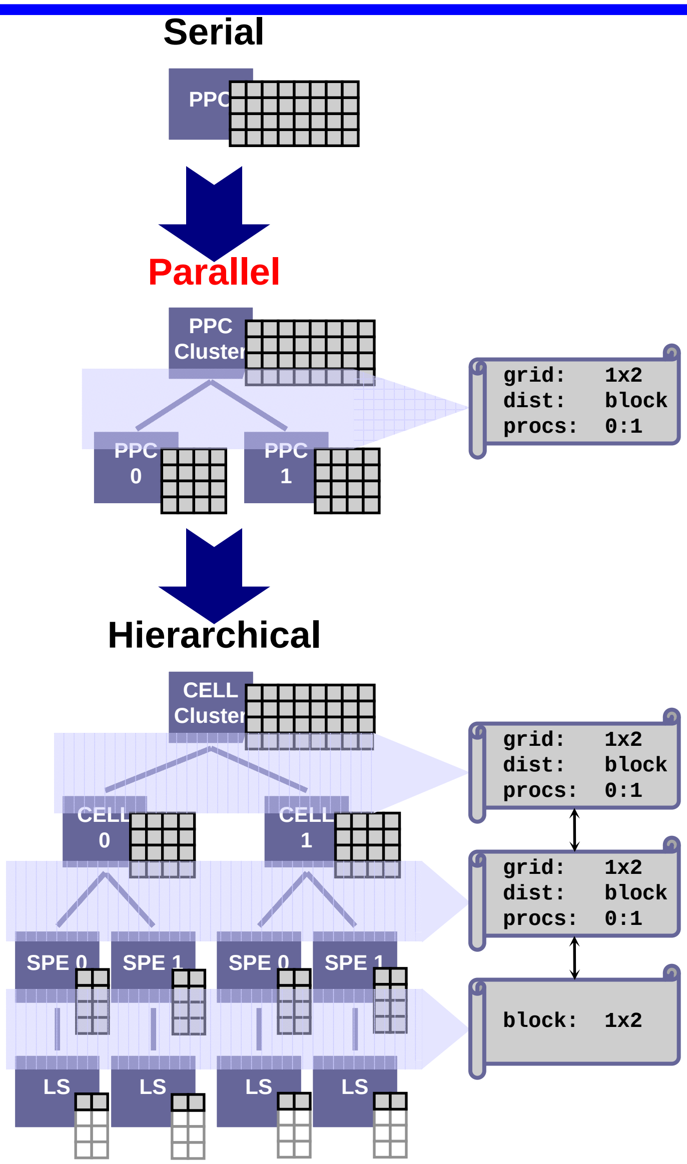
```
int main(int argc, char *argv[])
{
   PvtolProgram pvtol(argc, argv);










   // Allocate the array
   typedef Dense<2, int> BlockType;
   typedef Matrix<int, BlockType>             MatType;
   MatType matrix(4, 8);
}
```



Hierarchical Maps and Arrays

Example - Parallel

```
int main(int argc, char *argv[])
{
   PvtolProgram pvtol(argc, argv);










   // Distribute columns across 2 Cells


   Grid cellGrid(1, 2);
   DataDistDescription cellDist(BlockDist(0), BlockDist(0));
   RankList cellProcs(2);
   RuntimeMap cellMap(cellProcs, cellGrid, cellDist);

   // Allocate the array
   typedef Dense<2, int> BlockType;
   typedef Matrix<int, BlockType, RuntimeMap> MatType;
   MatType matrix(4, 8, cellMap);
}
```

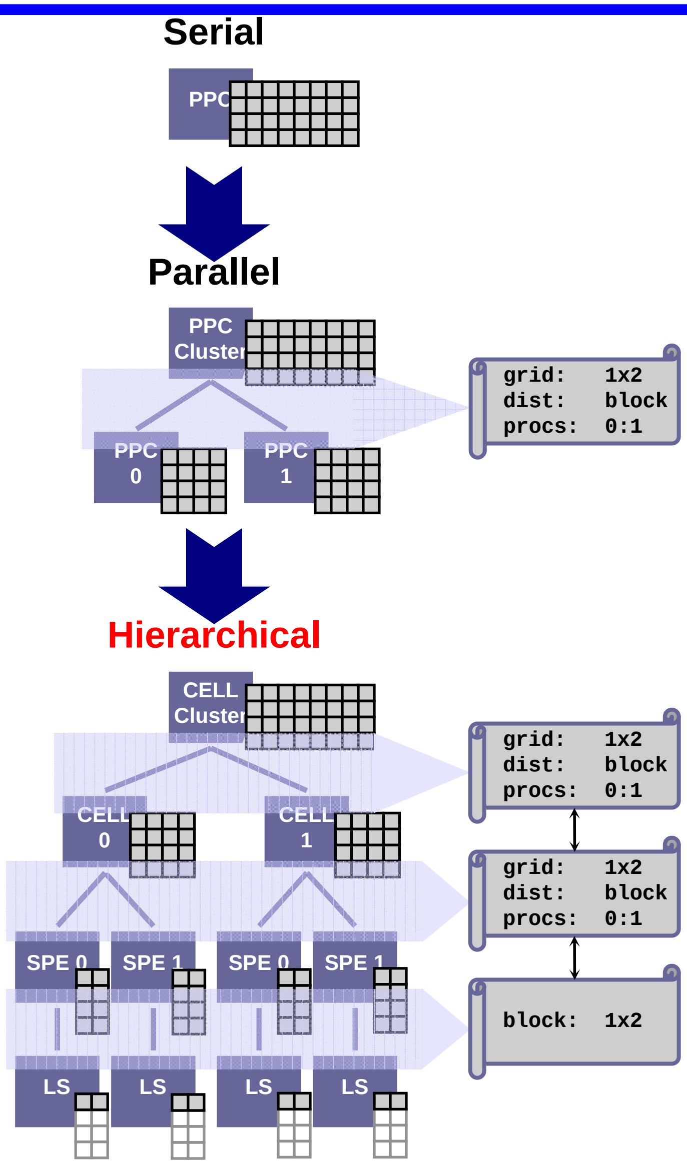


MIT Lincoln Laboratory

Hierarchical Maps and Arrays

Example - Hierarchical

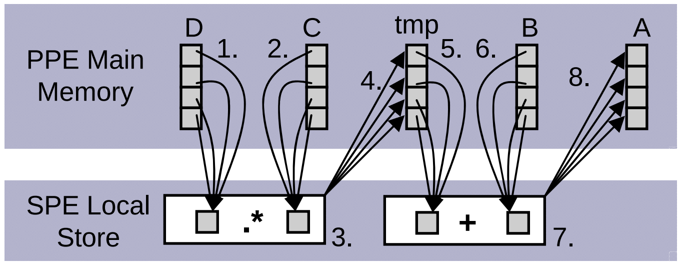
```
int main(int argc, char *argv[])
{
   PvtolProgram pvtol(argc, argv);

   // Distribute into 1x1 blocks
   unsigned int speLsBlockDims[2] = {1, 2};
   TemporalBlockingInfo speLsBlock(2, speLsBlockDims);
   TemporalMap speLsMap(speLsBlock);

   // Distribute columns across 2 SPEs
   Grid speGrid(1, 2);
   DataDistDescription speDist(BlockDist(0), BlockDist(0));
   RankList speProcs(2);
   RuntimeMap speMap(speProcs, speGrid, speDist, speLsMap);

   // Distribute columns across 2 Cells
   vector<RuntimeMap *> vectSpeMaps(1);
   vectSpeMaps.push_back(&speMap);
   Grid cellGrid(1, 2);
   DataDistDescription cellDist(BlockDist(0), BlockDist(0));
   RankList cellProcs(2);
   RuntimeMap cellMap(cellProcs, cellGrid, cellDist, vectSpeMaps);

   // Allocate the array
   typedef Dense<2, int> BlockType;
   typedef Matrix<int, BlockType, RuntimeMap> MatType;
   MatType matrix(4, 8, cellMap);
}
```



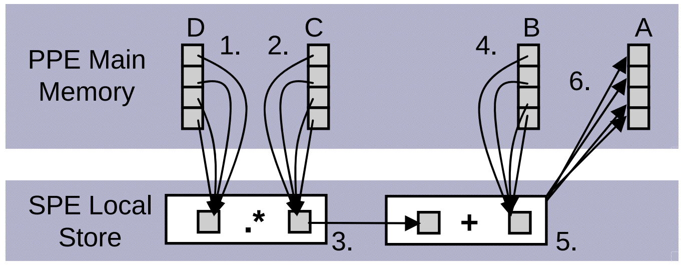
Functor Fusion

- **Expressions contain multiple operations**
  - **E.g. A = B + C .* D**
- **Functors encapsulate computation in objects**
- **Fusing functors improves performance by removing need for temporary variables**

Unfused



Fused



Let X_i be block i in array X

Unfused

Perform tmp = C .* D for all blocks:
1. Load D_i into SPE local store
2. Load C_i into SPE local store
3. Perform $tmp_i = C_i .* D_i$
4. Store tmp_i in main memory

Perform A = tmp + B for all blocks:
5. Load tmp_i into SPE local store
6. Load B_i into SPE local store
7. Perform $A_i = tmp_i + B_i$
8. Store A_i in main memory

Fused

Perform A = B + C .* D for all blocks:
1. Load D_i into SPE local store
2. Load C_i into SPE local store
3. Perform $tmp_i = C_i .* D_i$
4. Load B_i into SPE local store
5. Perform $A_i = tmp_i + B_i$
6. Store A_i in main memory

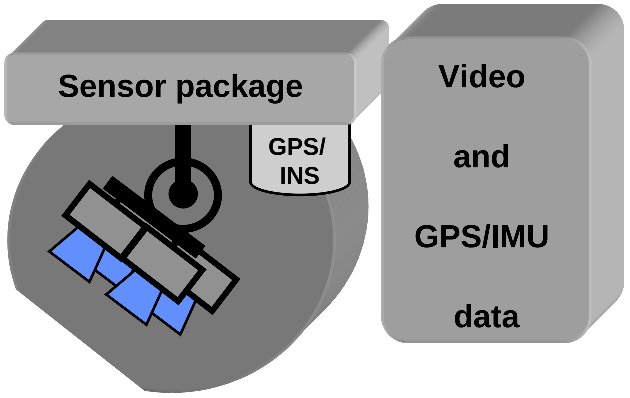
.* = elementwise multiplication

Outline

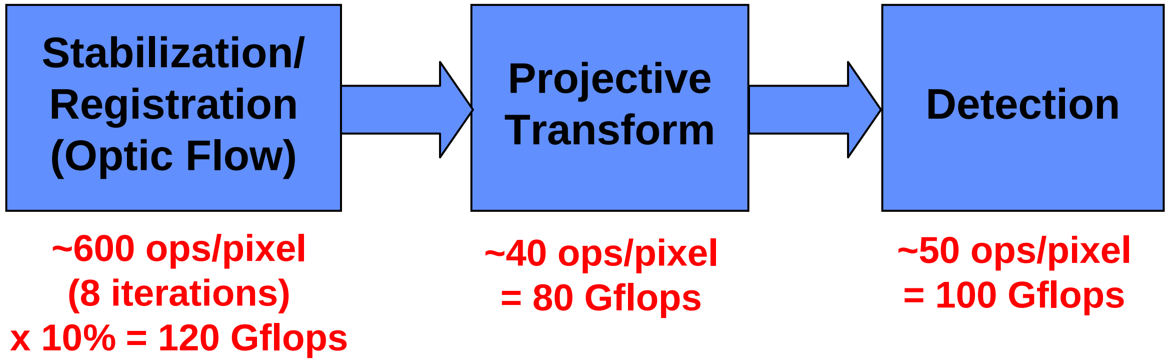
- Background
- Tasks & Conduits
- Maps & Arrays
- **Results**
- Summary

Persistent Surveillance

Canonical Front End Processing

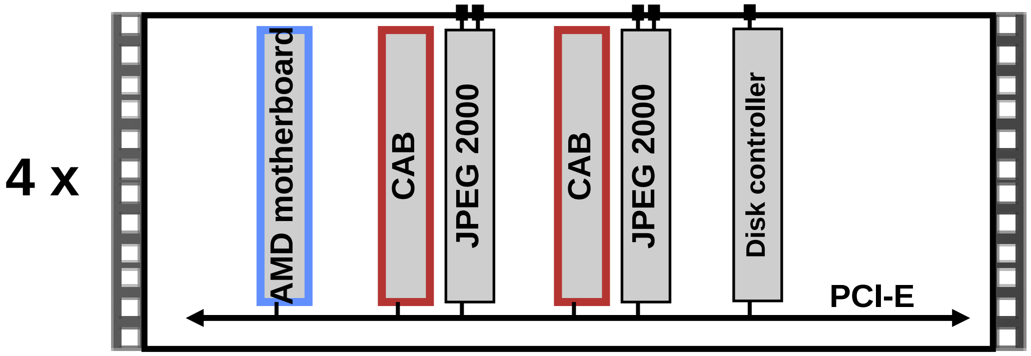


Processing Requirements ~300 Gflops


Stabilization/ Registration (Optic Flow)
Projective Transform
Detection
~600 ops/pixel (8 iterations) x 10% = 120 Gflops
~40 ops/pixel = 80 Gflops
~50 ops/pixel = 100 Gflops

4U Mercury Server
- 2 x AMD CPU motherboard
- 2 x Mercury Cell Accelerator Boards (CAB)
- 2 x JPEG 2000 boards
- PCI Express (PCI-E) bus

Logical Block Diagram



Signal and image processing turn sensor data into viewable images

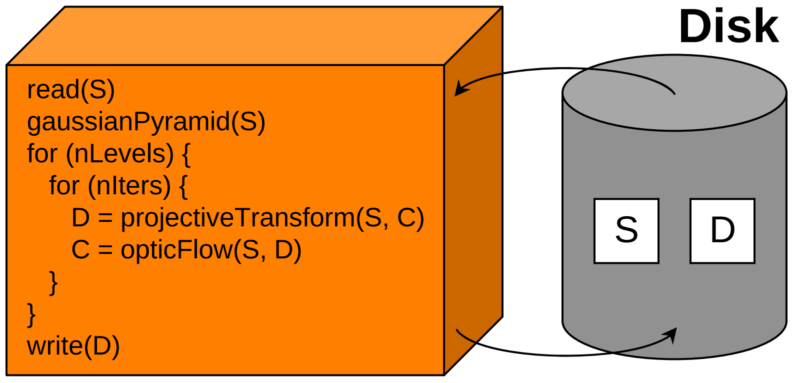
Post-Processing Software

Current CONOPS

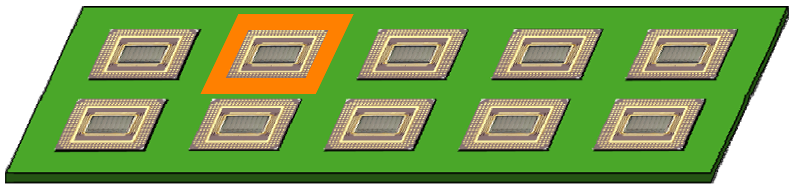
- Record video in-flight
- Apply registration and detection on the ground
- Analyze results on the ground

Future CONOPS

- Record video in-flight
- Apply registration and detection in-flight
- Analyze data on the ground



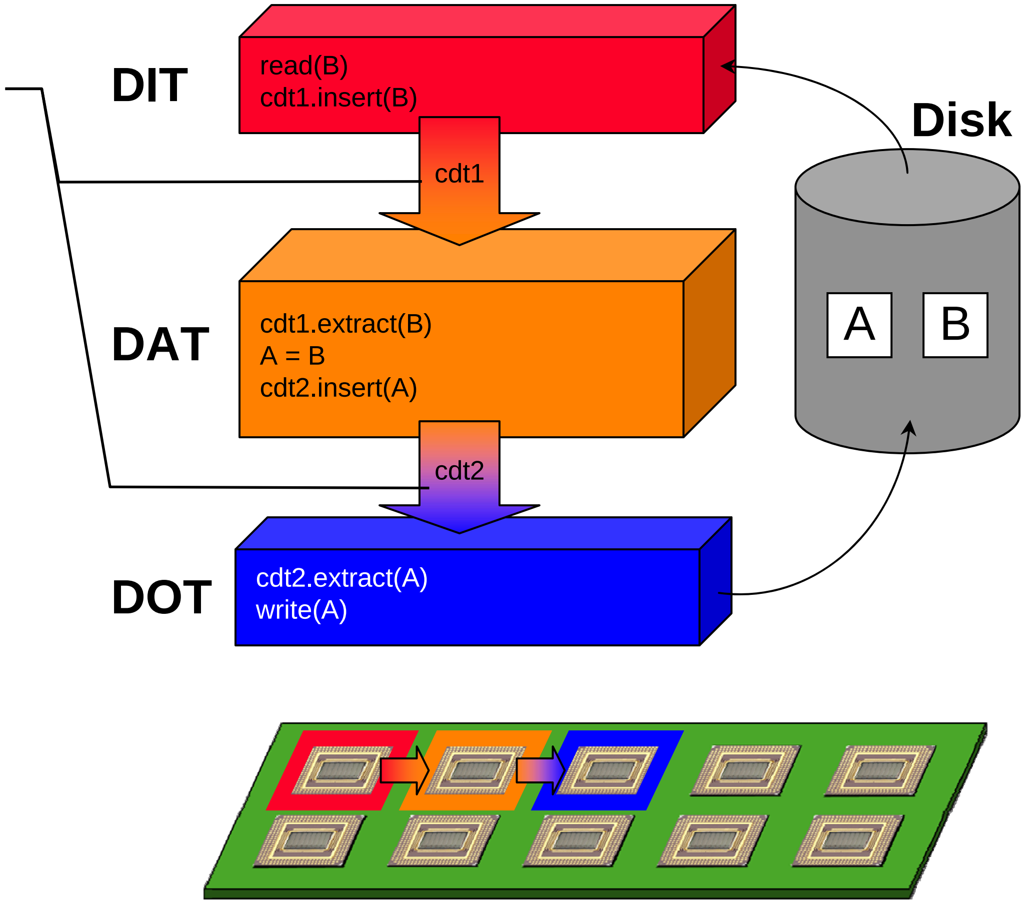
```
read(S)
gaussianPyramid(S)
for (nLevels) {
  for (nIters) {
    D = projectiveTransform(S, C)
    C = opticFlow(S, D)
  }
}
write(D)
```



Real-Time Processing Software

Step 1: Create skeleton DIT-DAT-DOT

Input and output of DAT should match input and output of application



Tasks and Conduits separate I/O from computation

* DIT – Data Input Task, DAT – Data Analysis Task, DOT – Data Output Task

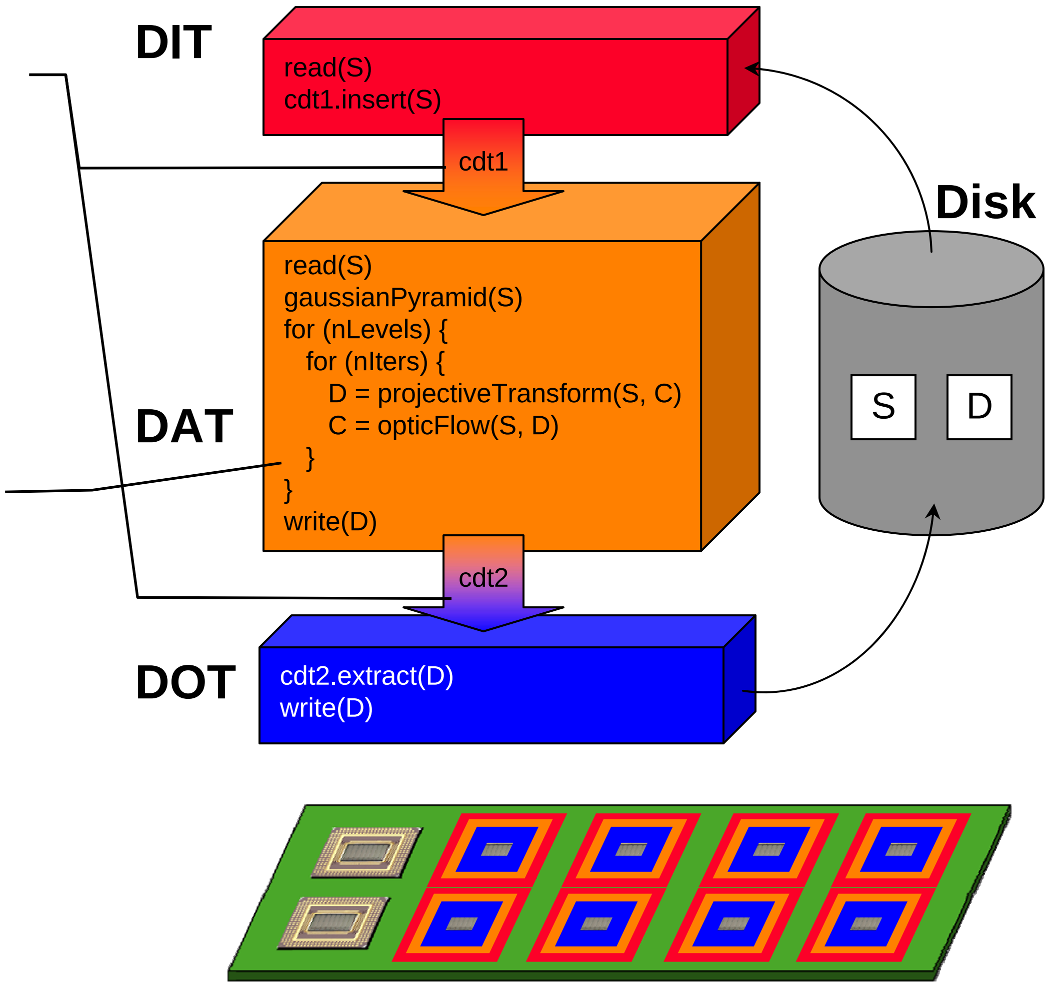
Real-Time Processing Software

Step 2: Integrate application code into DAT

Replace disk I/O with conduit reader and writer

Replace DAT with application code

Tasks and Conduits make it easy to change components



```
read(S)
cdt1.insert(S)
```

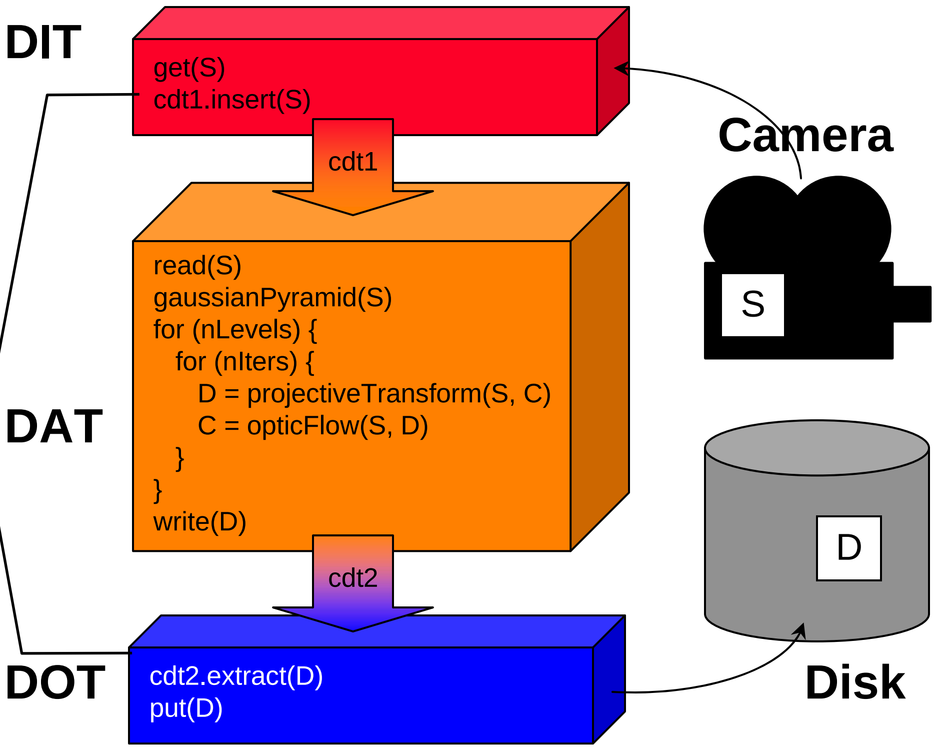
```
read(S)
gaussianPyramid(S)
for (nLevels) {
  for (nIters) {
    D = projectiveTransform(S, C)
    C = opticFlow(S, D)
  }
}
write(D)
```

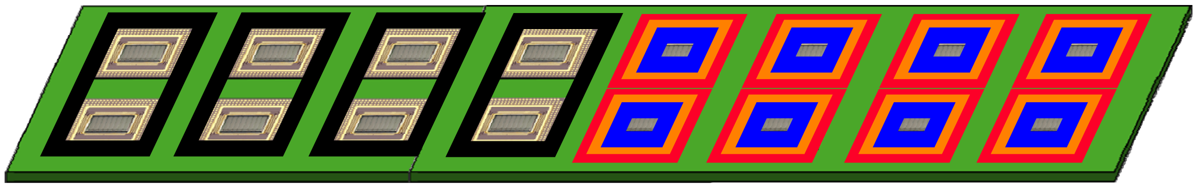
```
cdt2.extract(D)
write(D)
```

Real-Time Processing Software

Step 3: Replace disk with camera

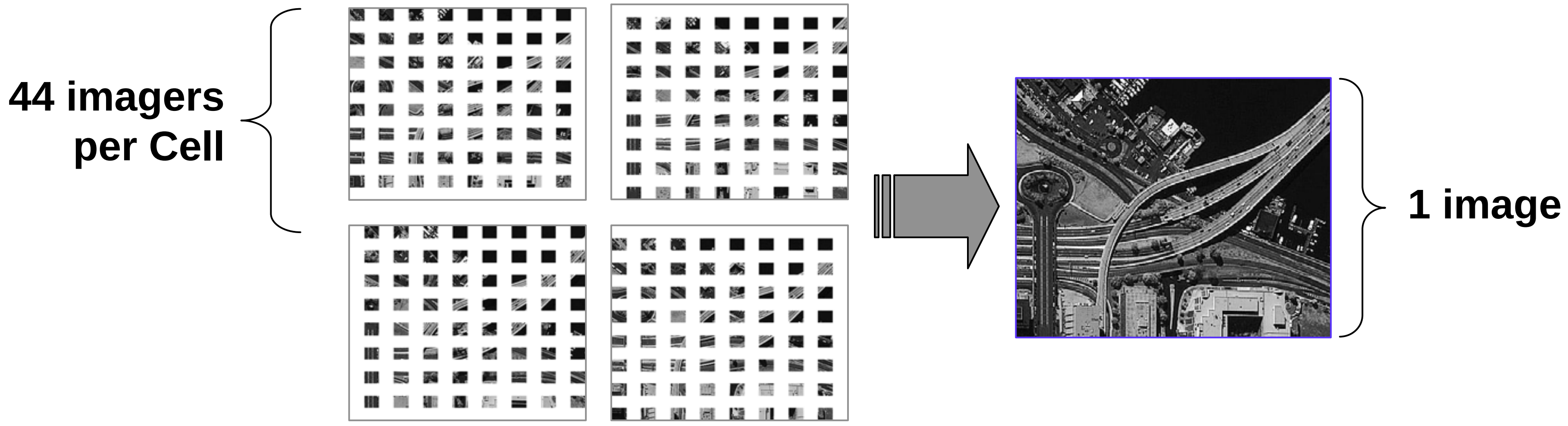
Replace disk I/O with bus I/O that retrieves data from the camera





MIT Lincoln Laboratory

Performance

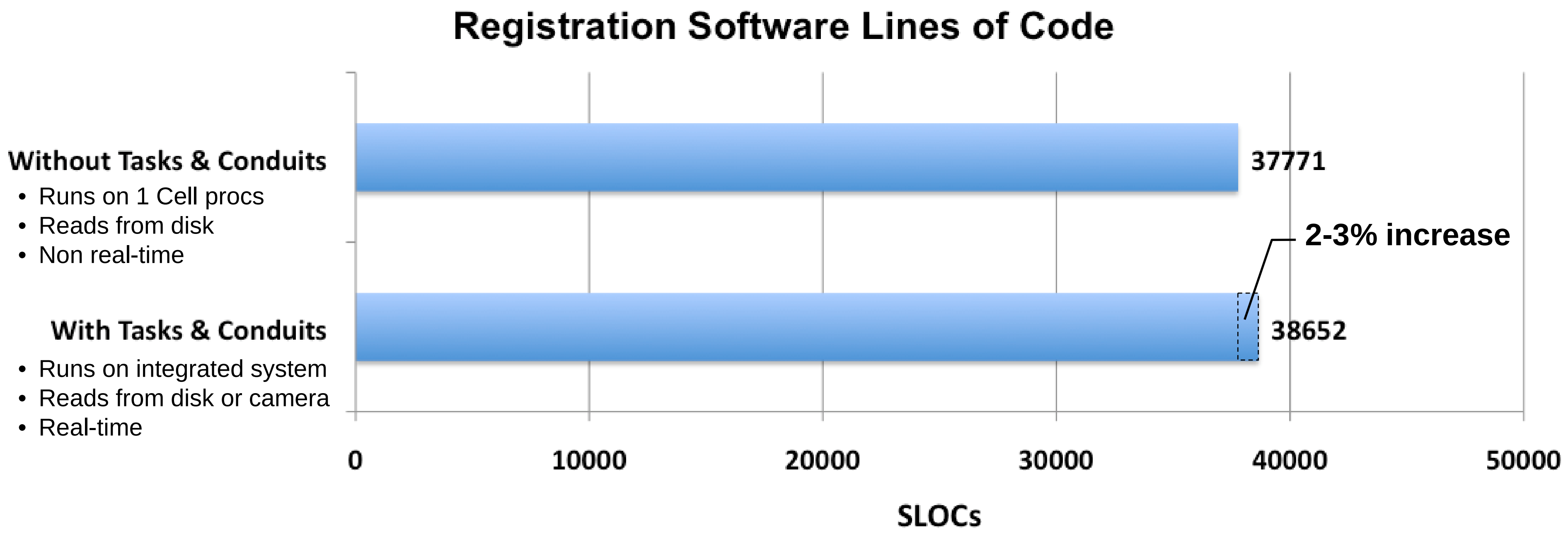


# imagers per Cell	Registration Time (w/o Tasks & Conduits)	Registration Time (w/ Tasks & Conduits*)
All imagers	1188 ms	1246 ms
1/2 of imagers	594 ms	623 ms
1/4 of imagers	297 ms	311 ms
Real-time Target Time	**500 ms**	**500 ms**

Tasks and Conduits incur little overhead

* Double-buffered

Performance vs. Effort



Benefits of Tasks & Conduits

- **Isolates I/O code from computation code**
  - **Can switch between disk I/O and camera I/O**
  - **Can create test jigs for computation code**
- **I/O and computation run concurrently**
  - **Can move I/O and computation to different processors**
  - **Can add multibuffering**

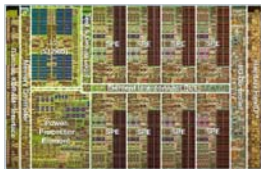
MIT Lincoln Laboratory

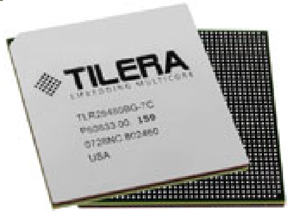
Outline

- Background
- Tasks & Conduits
- Hierarchical Maps & Arrays
- Results
- Summary

Future (Co-)Processor Trends

Multicore





IBM PowerXCell 8i
- 9 cores: 1 PPE + 8 SPE
- 204.8 GFLOPS single precision
- 102.4 GFLOPS double precision
- 92 W peak (est.)

Tilera TILE64
- 64 cores
- 443 GOPS
- 15 – 22 W @ 700 MHz

FPGAs

Xilinx Virtex-5
- Up to 330,000 logic cells
- 580 GMACS using DSP slices
- PPC 440 processor block

Curtis Wright CHAMP-FX2
- VPX-REDI
- 2 Xilinx Virtex-5 FPGAs
- Dual-core PPC 8641D

GPUs

NVIDIA Tesla C1060
- PCI-E x16
- ~1 TFLOPS single precision
- 225 W peak, 160 W typical

ATI FireStream 9250
- PCI-E x16
- ~1 TFLOPS single precision
- ~200 GFLOPS double precision
- 150 W

* Information obtained from manufacturers’ websites

Summary

- Modern DoD sensors have tight SWaP constraints
  - Multicore processors help achieve performance requirements within these constraints
- Multicore architectures are extremely difficult to program
  - Fundamentally changes the way programmers have to think
- PVTOL provides a simple means to program multicore processors
  - Refactored a post-processing application for real-time using Tasks & Conduits
  - No performance impact
  - Real-time application is modular and scalable
- We are actively developing PVTOL for Intel and Cell
  - Plan to expand to other technologies, e.g. FPGA's, automated mapping
  - Will propose to HPEC-SI for standardization

Acknowledgements

Persistent Surveillance Team

- Bill Ross
- Herb DaSilva
- Peter Boettcher
- Chris Bowen
- Cindy Fang
- Imran Khan
- Fred Knight
- Gary Long
- Bobby Ren

PVTOL Team

- Bob Bond
- Nayda Bliss
- Karen Eng
- Jeremiah Gale
- James Geraci
- Ryan Haney
- Jeremy Kepner
- Sanjeev Mohindra
- Sharon Sacco
- Eddie Rutledge