THE UNIVERSITY OF TEXAS AT AUSTIN

Sequoia

Mattan Erez

The University of Texas at Austin

Emerging Themes

- Writing high-performance code amounts to…
  - Intelligently structuring algorithms
    [compiler help unlikely]
  - Efficiently using communication
  - Efficiently using parallel resources
    [compilers struggle without help]
  - Generating efficient inner loops (kernels)
    [compilers coming around]

Sequoia's goals

- Facilitate development of hierarchy-aware stream programs …
- … that remain portable across machines

- Provide constructs that can be implemented efficiently without requiring advanced compiler technology (but facilitate optimization)
  - Place computation and data in machine
  - Explicit parallelism and communication
  - Large bulk transfers

- Get out of the way when needed

Sequoia

- Language: generalized stream programming for machines with deep memory hierarchies
- Idea: Expose abstract memory hierarchy to programmer
- Implementation: **language, compiler, tuner, and runtime**
  - benchmarks run well on Cell processor based systems, clusters of PCs, SMPs, out-of-core computation, and combinations of above

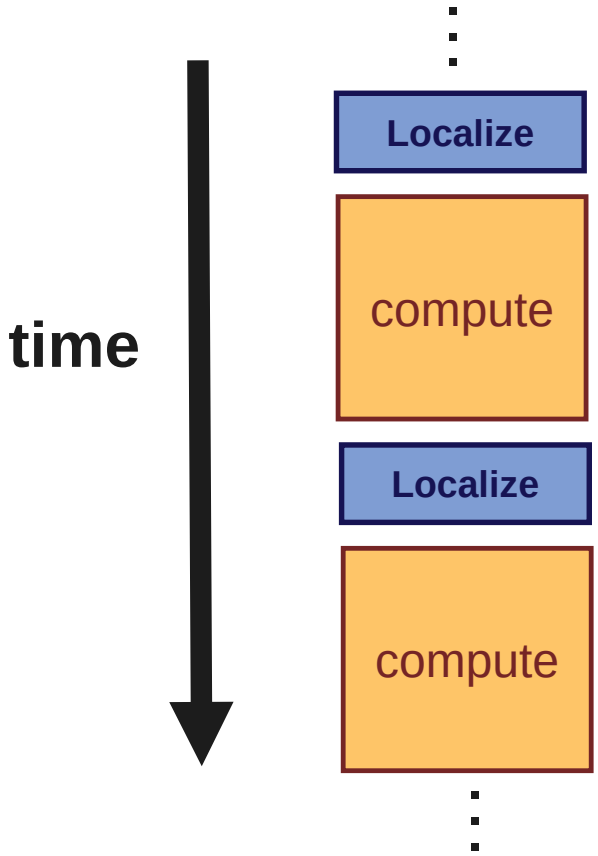
- Key challenge in high performance programming is:

- **communication (not parallelism)**

  - **Latency**
  - **Bandwidth**

Avoiding latency stalls

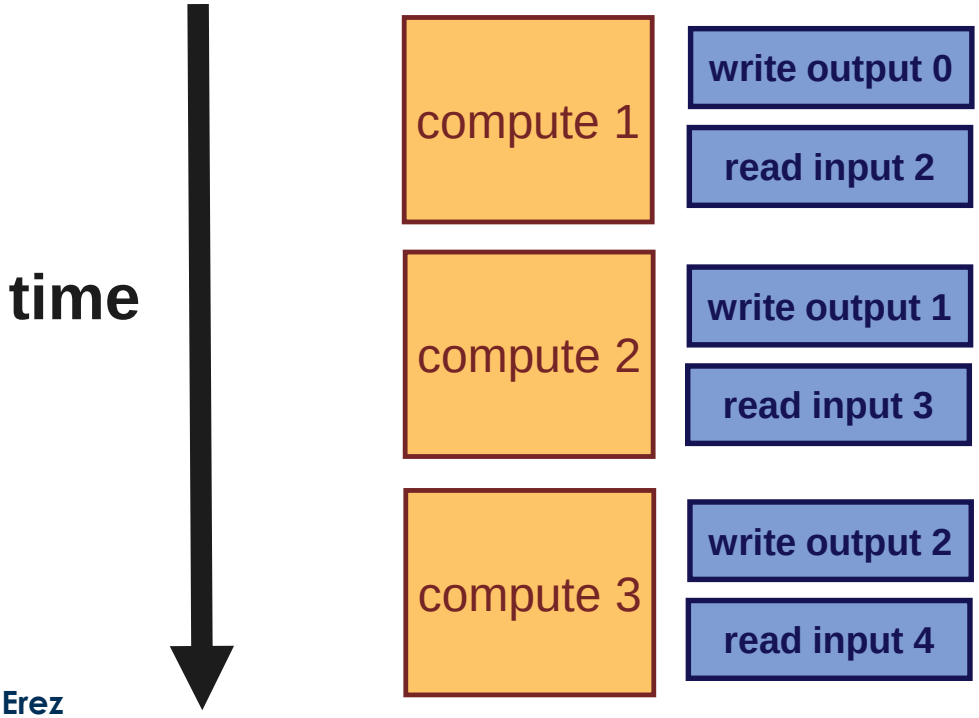
- Exploit locality to minimize number of stalls
  - Example: Blocking / tiling



Avoiding latency stalls

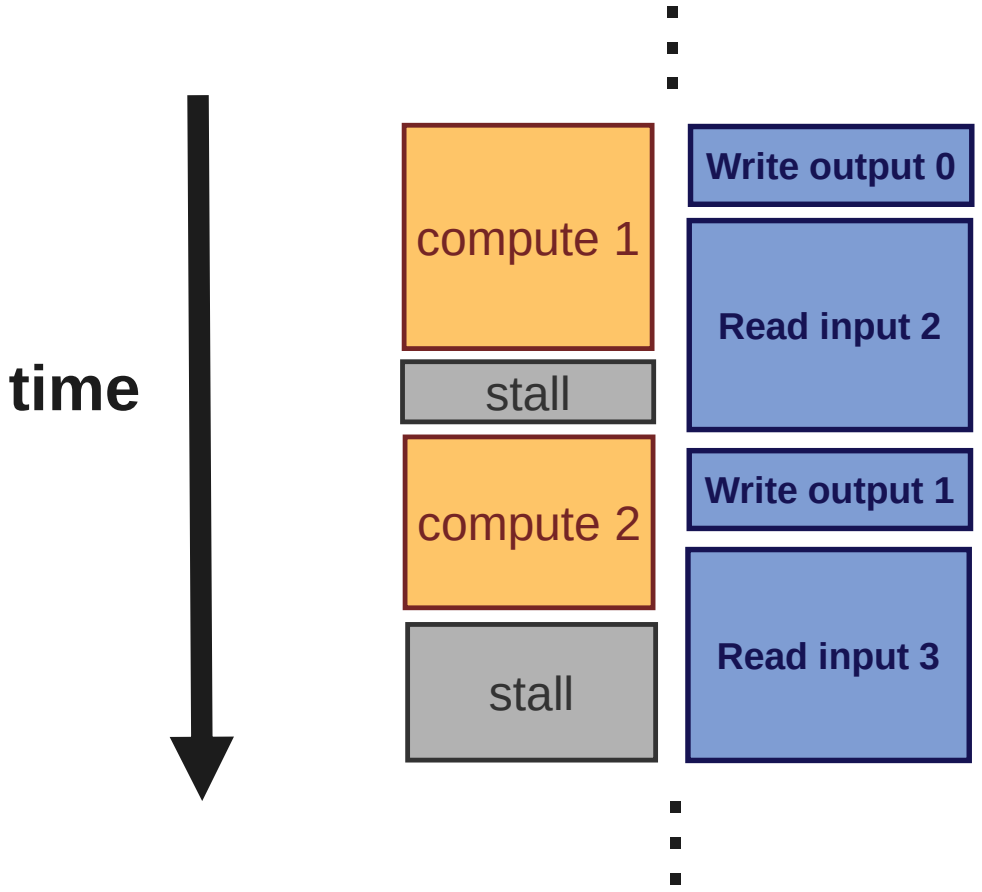
1. Prefetch batch of data
2. Compute on data (avoiding stalls)
3. Initiate write of results

- … Then compute on next batch (which should be loaded)



Exploit locality

- Compute > bandwidth, else execution stalls



Streaming

- Streaming involves structuring algorithms as collections of independent [locality cognizant] computations with well-defined working sets.

- **This structuring may be done at any scale.**

Keep temporaries in registers

Cache/scratchpad blocking

Message passing on a cluster

Out-of-core algorithms

Locality in programming languages

- Local (private) vs. global (remote) addresses
  - UPC, Titanium
- Domain distributions (map array elements to location)
  - HPF, UPC, ZPL
  - Adopted by DARPA HPCS: X10, Fortress, Chapel

Focus on communication between nodes

Ignore hierarchy within a node

Locality in programming languages

- Streams and kernels
  - Stream data off chip. Kernel data on chip.
  - StreamC/KernelC, Brook
  - GPU shading (Cg, HLSL)
  - CUDA

Architecture specific

Only represent two levels

Hierarchy-aware models

- Cache obliviousness (recursion)

- Space-limited procedures (Alpern et al.)

Programming methodologies, not programming environments

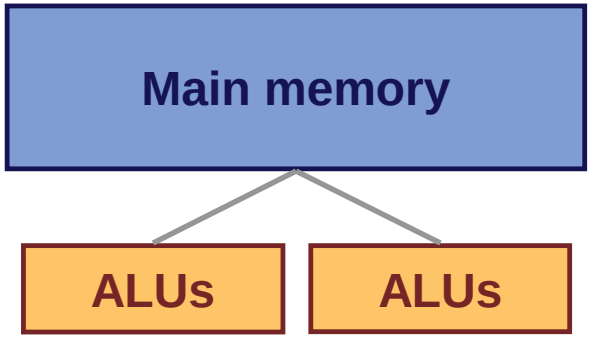
Role of programming model

- **Encourage hardware-friendly structure**
- Bulk operations
- Bandwidth matters: structure code to maximize locality
- Parallelism matters: make parallelism explicit
- Awareness of memory hierarchy applies everywhere
  - Keep temporaries in registers
  - Cache/scratchpad blocking
  - Message passing on a cluster
  - Out-of-core algorithms

Hierarchical memory in Sequoia

Hierarchical memory

- Abstract machines as trees of memories

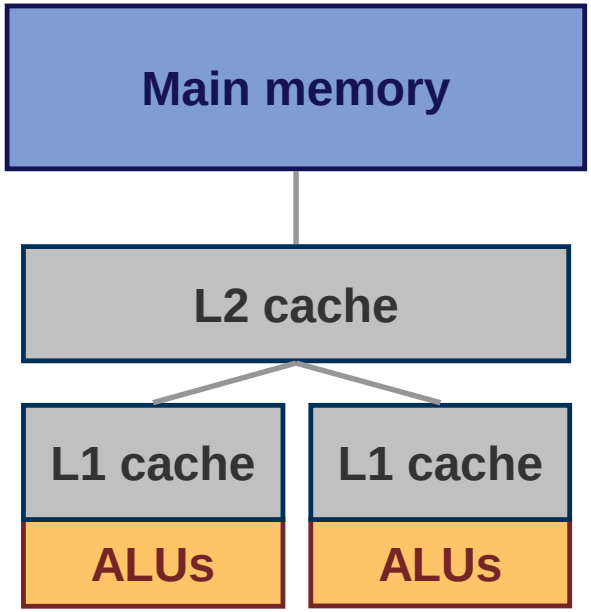


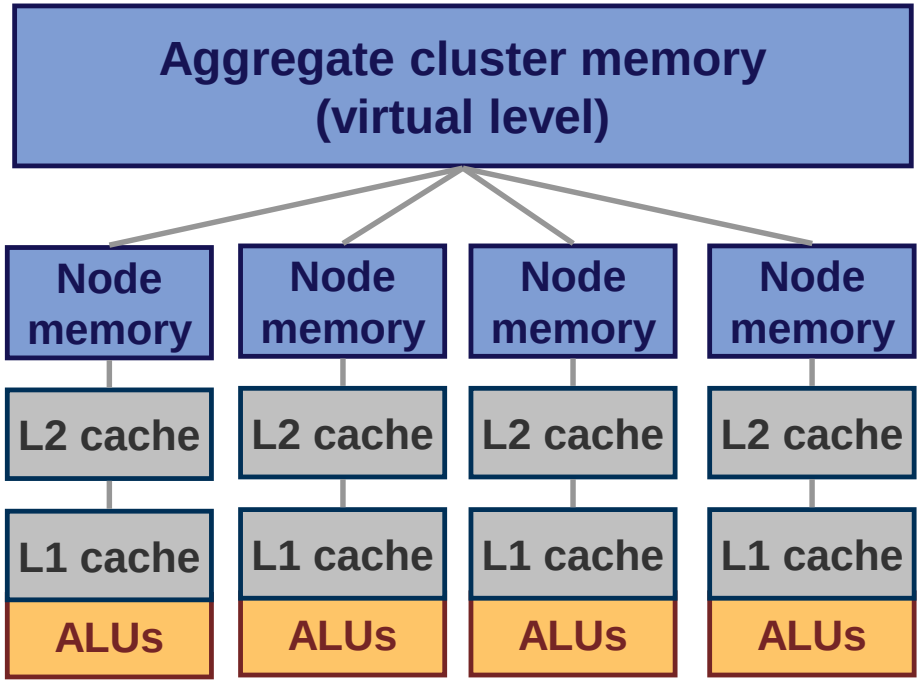
Similar to:

Parallel Memory Hierarchy Model (Alpern et al.)

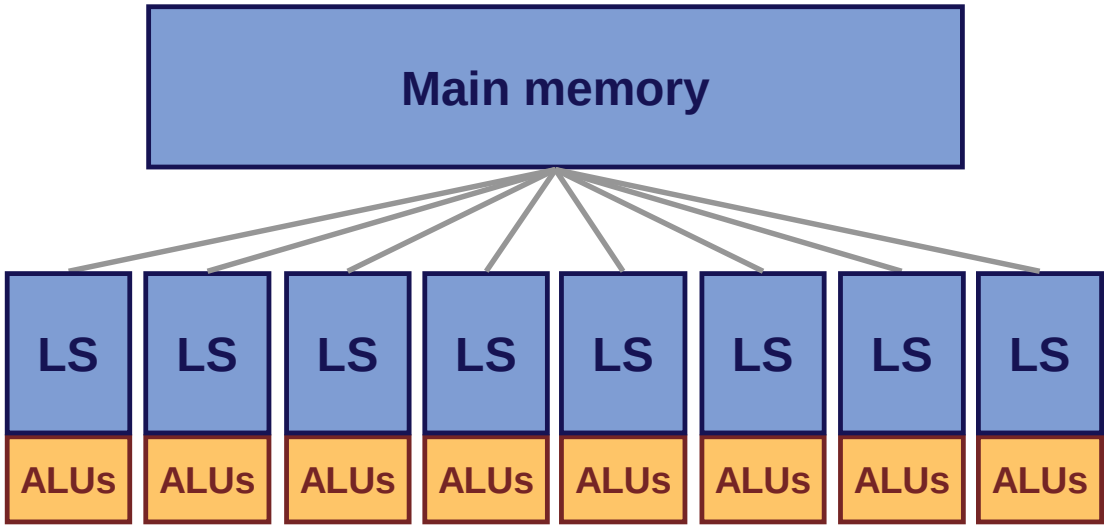
Hierarchical memory

- Abstract machines as trees of memories

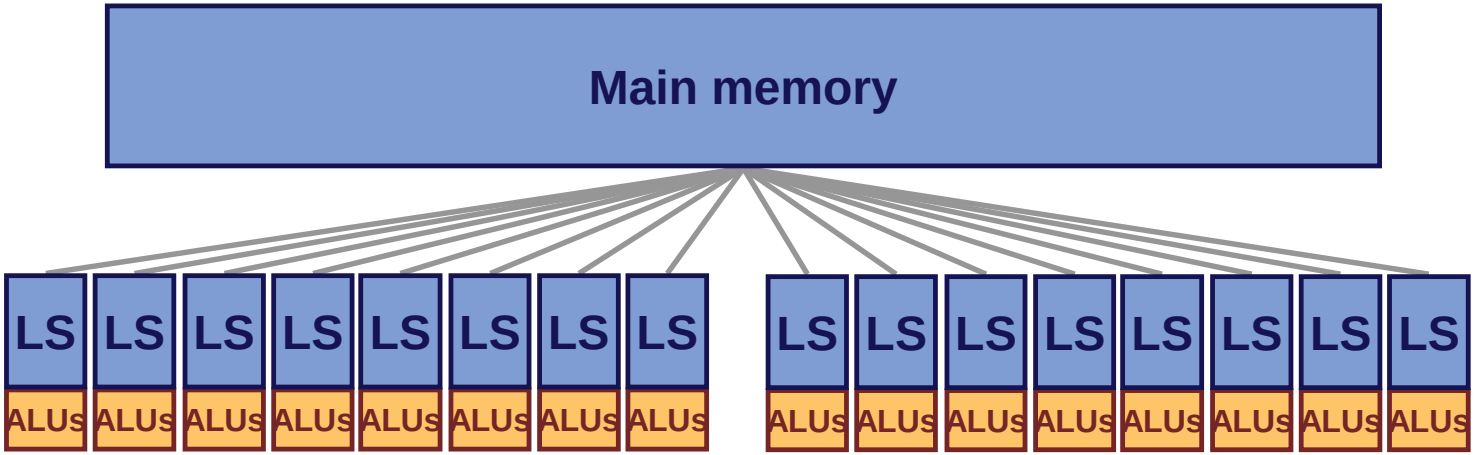




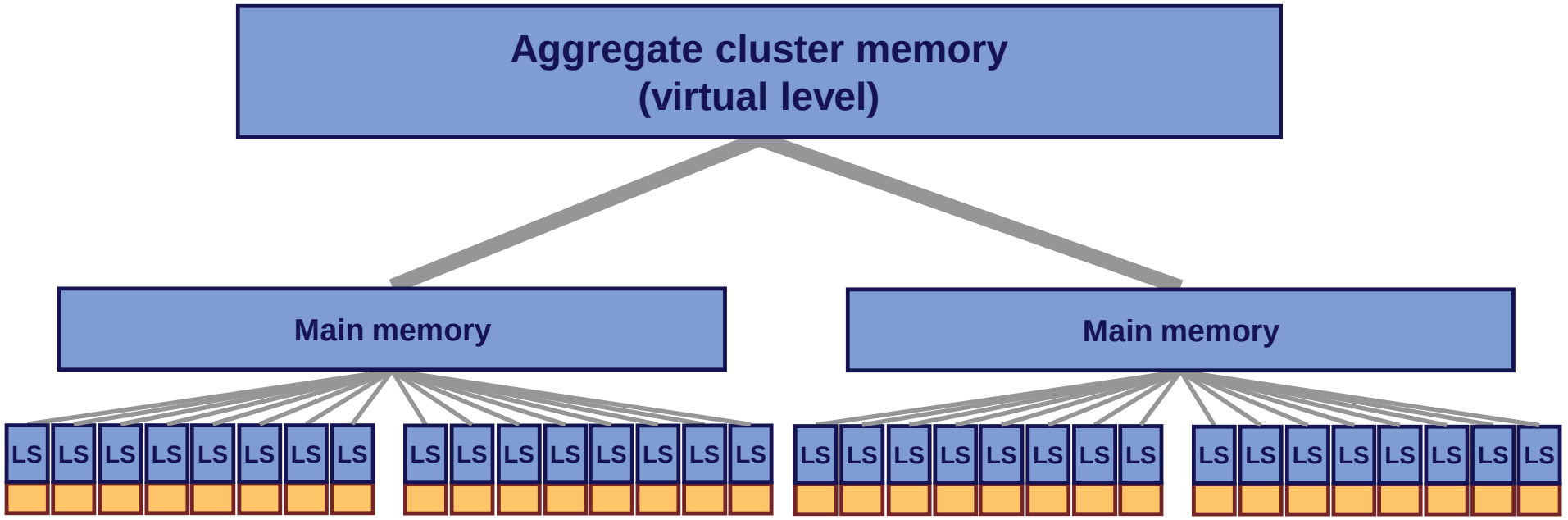
Hierarchical memory



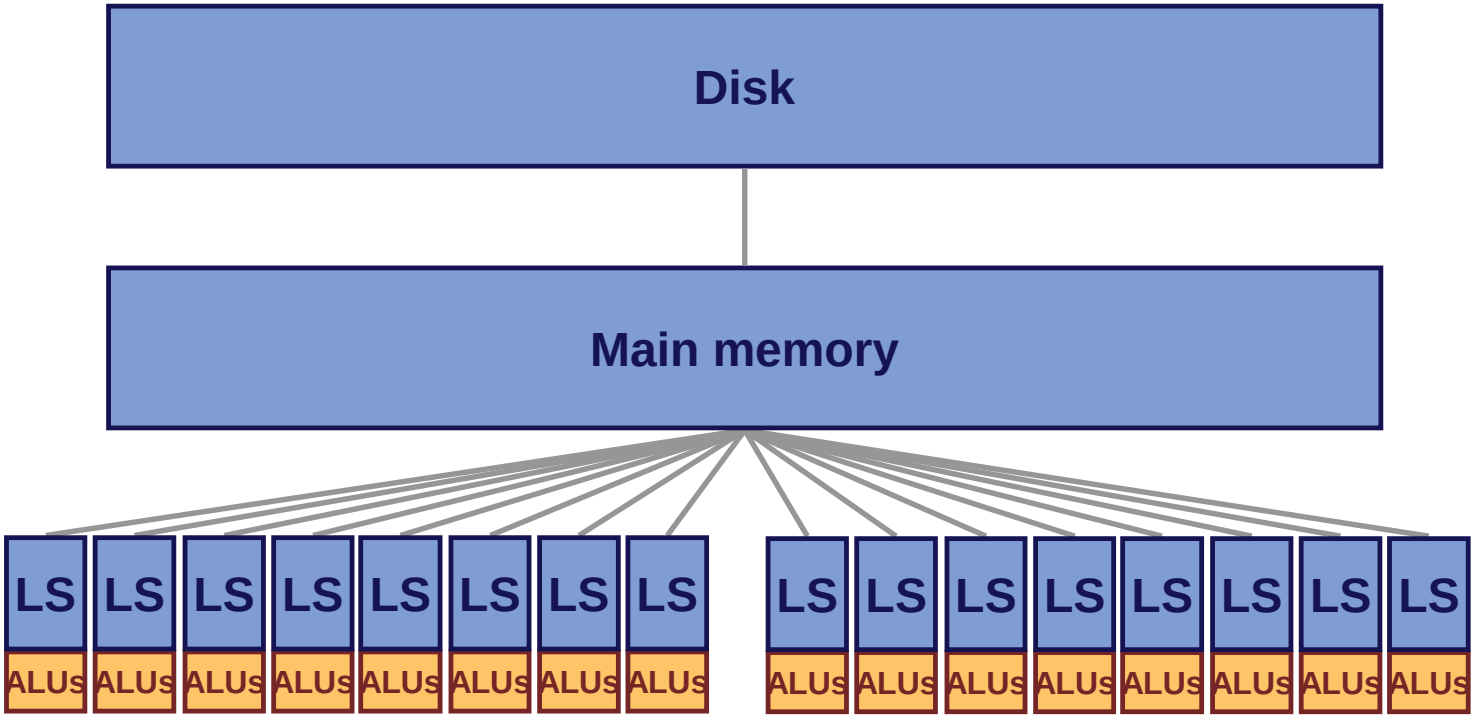
Hierarchical memory



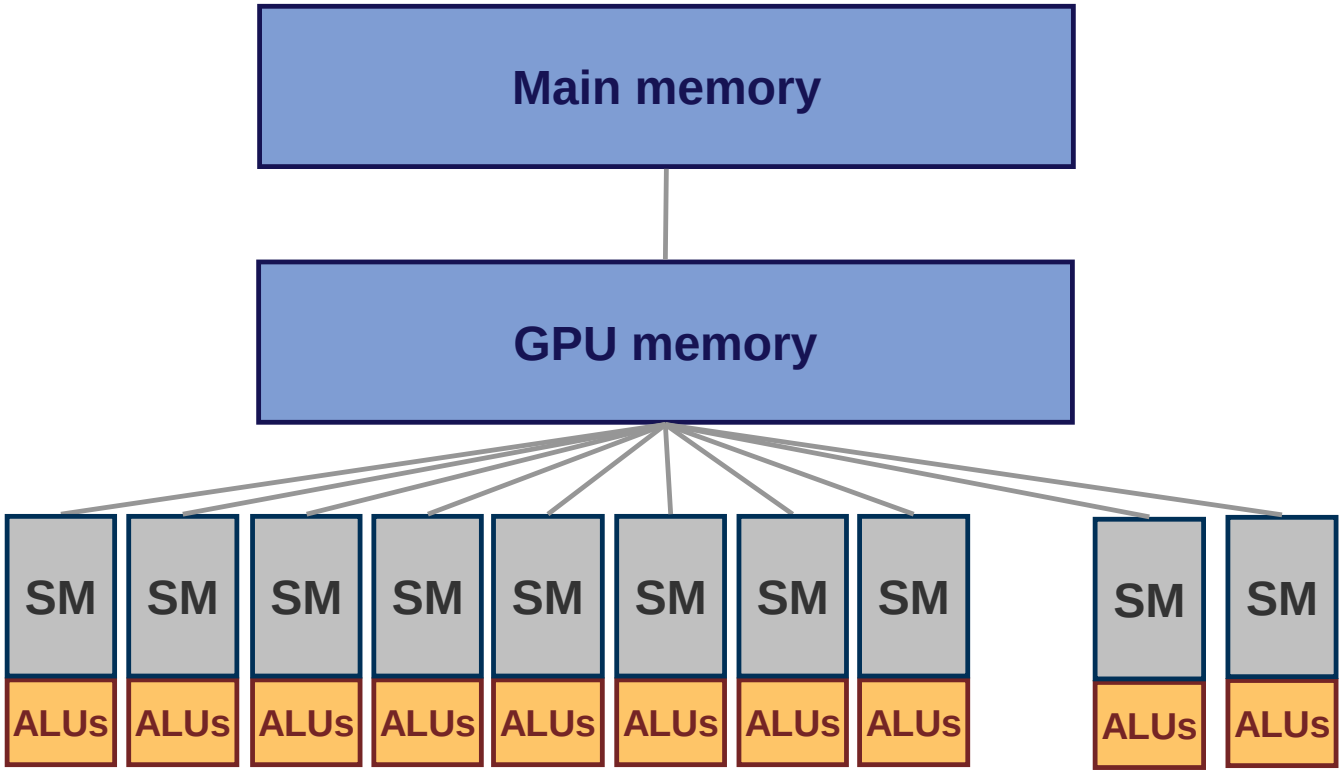
Hierarchical memory



Hierarchical memory



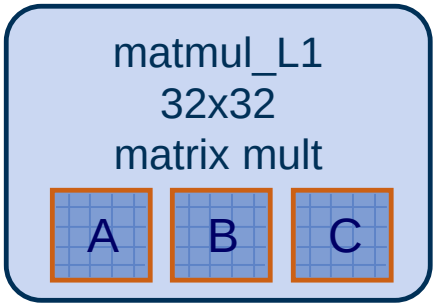
Hierarchical memory



Blocked matrix multiplication

C += A x B

```
void matmul_L1( int M, int N, int T,
                float* A,
                float* B,
                float* C)
{
   for (int i=0; i<M; i++)
     for (int j=0; j<N; j++)
       for (int k=0; k<T; k++)
          C[i][j] += A[i][k] * B[k][j];
}
```



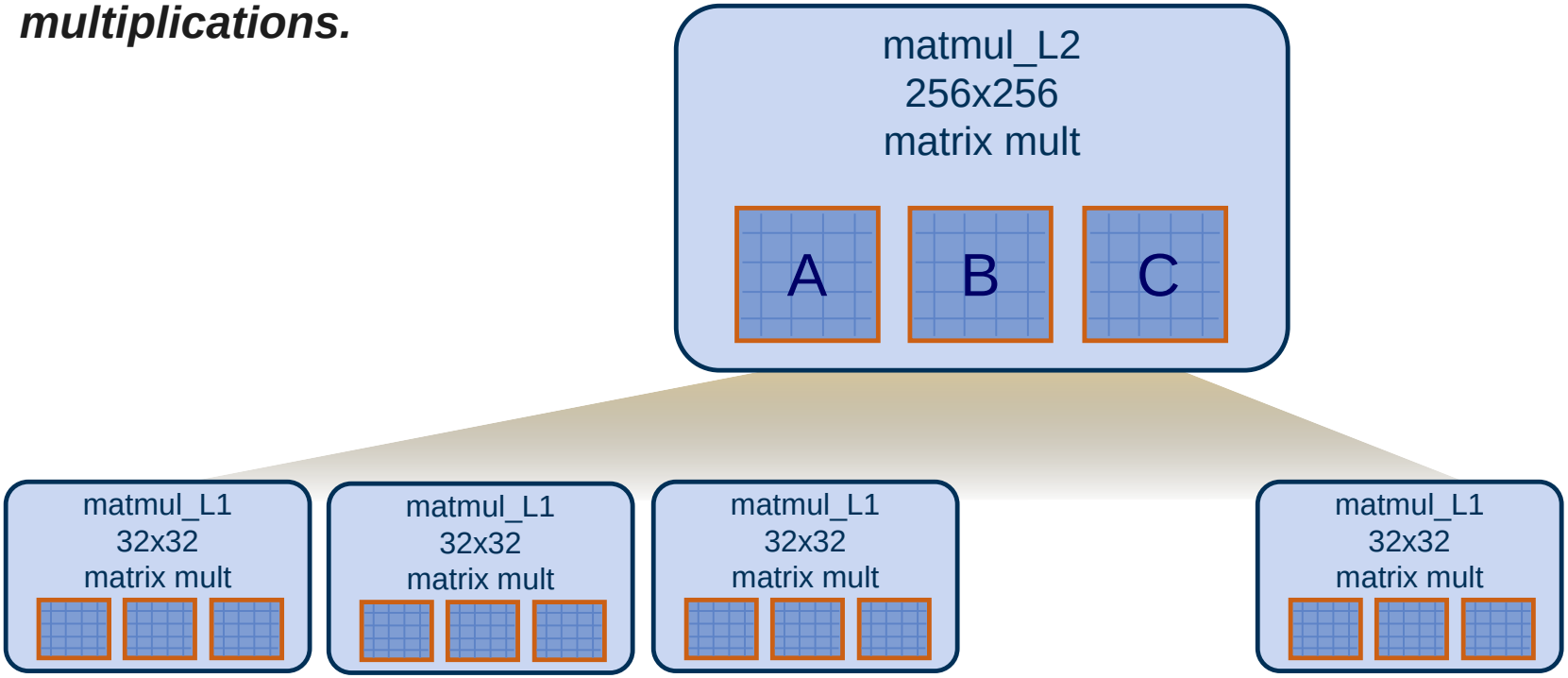
Blocked matrix multiplication

C += A x B

```
void matmul_L2( int M, int N, int T,
                float* A,
                float* B,
                float* C)
{
```

Perform series of L1 matrix multiplications.

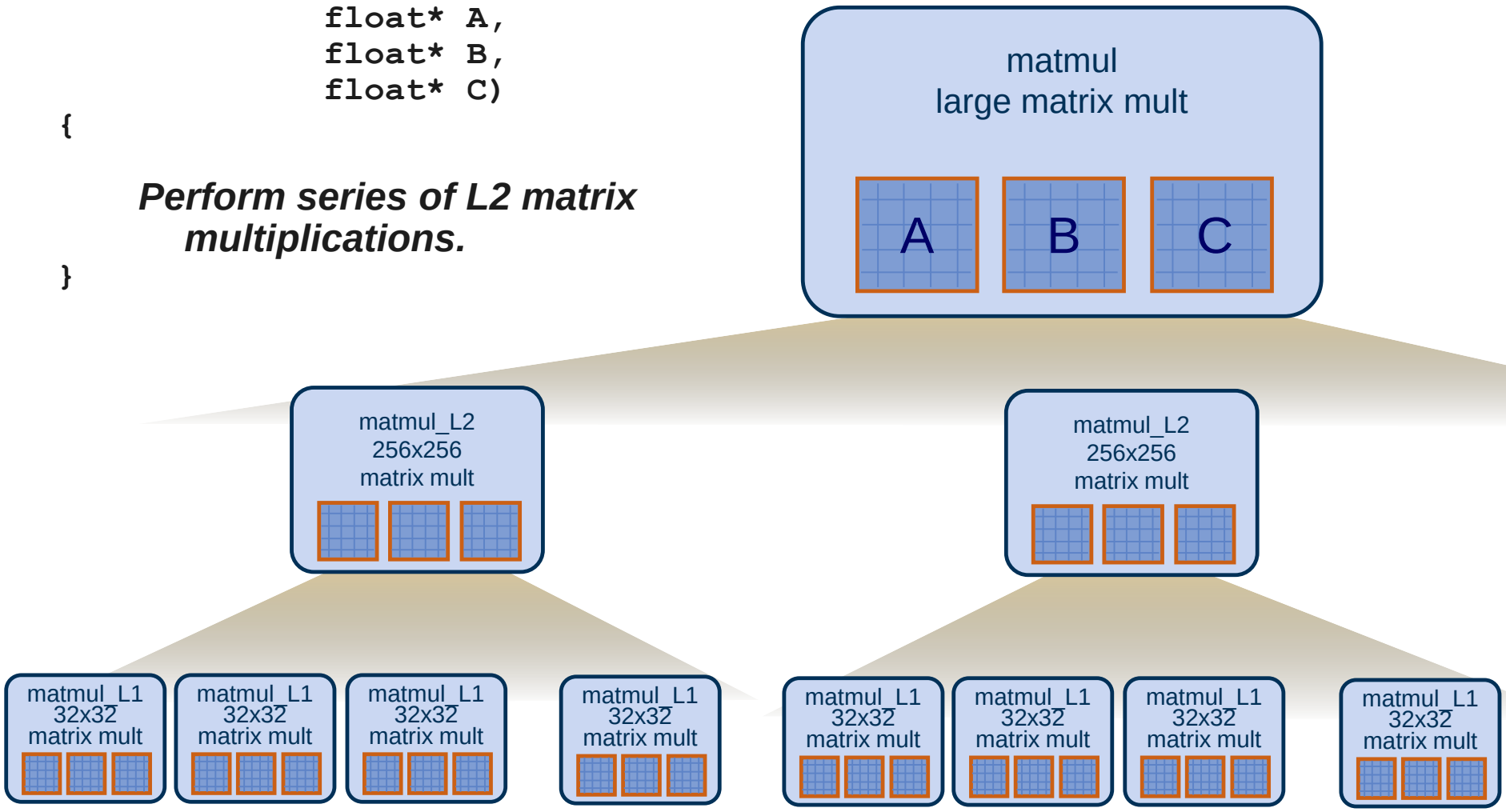
```
}
```



Blocked matrix multiplication

C += A x B

```
void matmul( int M, int N, int T,
             float* A,
             float* B,
             float* C)
{

    Perform series of L2 matrix
      multiplications.

}
```



Sequoia's method

- Explicit communication between abstract memories
- Locality awareness
- Hierarchy portability
  - Across machines, within levels of a machine
- **Programmer expresses combined computation and decomposition parameterized algorithm**
  - **System follows algorithm to map to a specific machine**

Sequoia tasks

Sequoia tasks

- Special functions called **tasks** are the building blocks of Sequoia programs

```
task matmul::leaf( in    float A[M][T],
                   in    float B[T][N],
                   inout float C[M][N]  )
{
   for (int i=0; i<M; i++)
     for (int j=0; j<N; j++)
       for (int k=0; k<T; k++)
          C[i][j] += A[i][k] * B[k][j];
}
```

Read-only parameters M, N, T give sizes of multidimensional arrays when task is called.

Sequoia tasks

- Single abstraction for
  - Isolation / parallelism
  - Explicit communication / working sets
  - Expressing locality

- Tasks operate on arrays, not array elements

- Tasks nest: they call subtasks

Sequoia tasks

- Task arguments and temporaries define a working set
- **Task working set resident at single location in abstract machine tree**

```
task matmul::leaf( in    float A[M][T],
                   in    float B[T][N],
                   inout float C[M][N] )
{
    for (int i=0; i<M; i++)
      for (int j=0; j<N; j++)
        for (int k=0; k<T; k++)
           C[i][j] += A[i][k] * B[k][j];
}
```

```
task matmul::inner( in    float A[M][T],
                    in    float B[T][N],
                    inout float C[M][N] )
{
  tunable int P, Q, R;
```

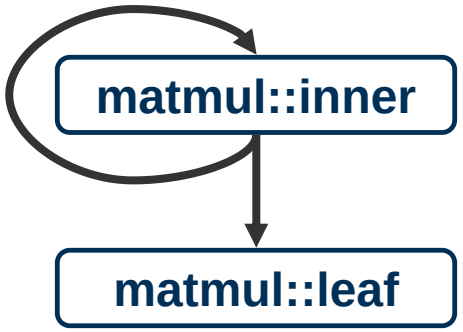
Recursively call `matmul` task on submatrices of A, B, and C of size PxQ, QxR, and PxR.

```
}
```

```
task matmul::leaf( in    float A[M][T],
                   in    float B[T][N],
                   inout float C[M][N]  )
{
   for (int i=0; i<M; i++)
     for (int j=0; j<N; j++)
       for (int k=0; k<T; k++)
          C[i][j] += A[i][k] * B[k][j];
}
```

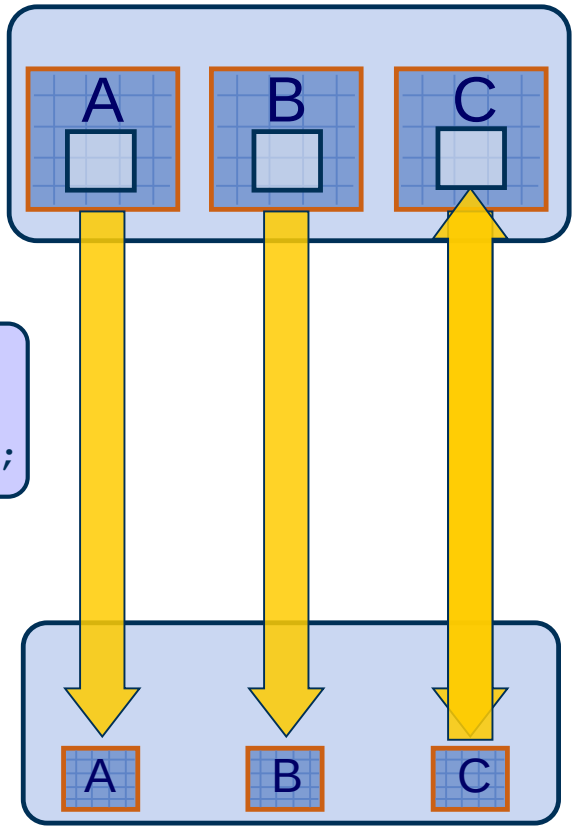
```
task matmul::inner( in    float A[M][T],
                    in    float B[T][N],
                    inout float C[M][N] )
{
  tunable int P, Q, R;

  mappar( int i=0 to M/P,
          int j=0 to N/R ) {
     mapseq( int k=0 to T/Q ) {

        matmul( A[P*i:P*(i+1);P][Q*k:Q*(k+1);Q],
                B[Q*k:Q*(k+1);Q][R*j:R*(j+1);R],
                C[P*i:P*(i+1);P][R*j:R*(j+1);R] );
     }
  }
}

task matmul::leaf( in    float A[M][T],
                   in    float B[T][N],
                   inout float C[M][N]  )
{
   for (int i=0; i<M; i++)
     for (int j=0; j<N; j++)
       for (int k=0; k<T; k++)
          C[i][j] += A[i][k] * B[k][j];
}
```

Variant call graph



```
task matmul::inner( in    float A[M][T],
                    in    float B[T][N],
                    inout float C[M][N] )
{
  tunable int P, Q, R;

  mappar( int i=0 to M/P,
          int j=0 to N/R ) {
     mapseq( int k=0 to T/Q ) {

        matmul( A[P*i:P*(i+1);P][Q*k:Q*(k+1);Q],
                B[Q*k:Q*(k+1);Q][R*j:R*(j+1);R],
                C[P*i:P*(i+1);P][R*j:R*(j+1);R] );
     }
  }
}

task matmul::leaf( in    float A[M][T],
                   in    float B[T][N],
                   inout float C[M][N]  )
{
   for (int i=0; i<M; i++)
     for (int j=0; j<N; j++)
       for (int k=0; k<T; k++)
          C[i][j] += A[i][k] * B[k][j];
}
```

Calling task: matmul::inner
Located at level *X*



Callee task: matmul::leaf
Located at level *Y*

```
task matmul::inner( in    float A[M][T],
                    in    float B[T][N],
                    inout float C[M][N] )
{
  tunable int P, Q, R;

  mappar( int i=0 to M/P,
          int j=0 to N/R ) {
    mapseq( int k=0 to T/Q ) {

      matmul( A[P*i:P*(i+1);P][Q*k:Q*(k+1);Q],
              B[Q*k:Q*(k+1);Q][R*j:R*(j+1);R],
              C[P*i:P*(i+1);P][R*j:R*(j+1);R] );
    }
  }
}
```

Leaf variants

- Be practical: Can use platform-specific kernels

```
task matmul::leaf(in    float A[M][T],
                  in    float B[T][N],
                  inout float C[M][N])
{
   for (int i=0; i<M; i++)
     for (int j=0; j<N; j++)
       for (int k=0;k<T; k++)
          C[i][j] += A[i][k] * B[k][j];
}
```

```
task matmul::leaf_cblas(in    float A[M][T],
                        in    float B[T][N],
                        inout float C[M][N])
{
  cblas_sgemm(A, M, T, B, T, N, C, M, N);
}
```

Synchronization

- *mapseq* implies sync at end of every iteration
- *mappar* implies sync at end of iteration space

- No explicit synchronization

- Synchronization is the trickiest part of parallel programming and one of the least portable
  - Help the user by structuring sync and allowing compiler to optimize the mechanism

- Bulk synchronous / Sequoia more robust than interacting threads

Synchronization Impacts Parallelism

- Parallelism explicitly expressed using *mappar*
  - DLP
- What about ILP?
  - Parallelism can exist within a leaf
    - Ignored by Sequoia but potential for ILP and SIMD
- What about TLP?
  - Implicit in dependence of operations
  - Allows pipeline parallelism within a mappar

Summary: Sequoia tasks

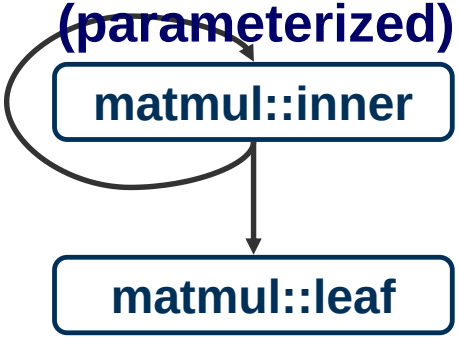
- Single abstraction for
  - Isolation / parallelism
  - Explicit communication / working sets
  - Expressing locality

- Sequoia programs describe hierarchies of tasks
  - Mapped onto memory hierarchy
  - Parameterized for portability
  - **Algorithm for decomposition**

Mapping tasks to machines

How mapping works

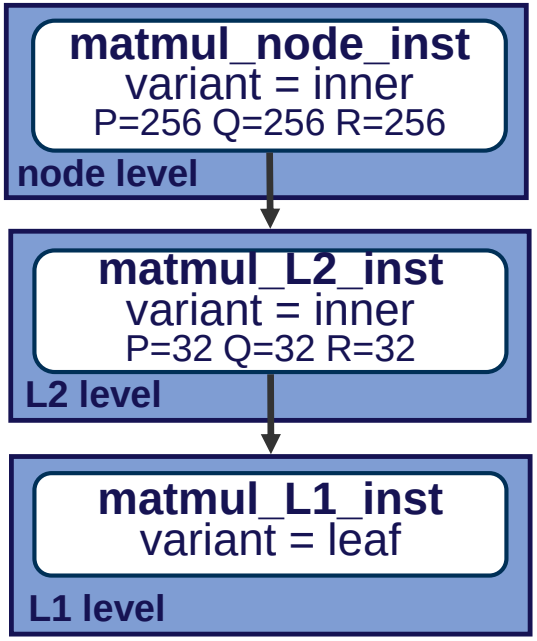
Sequoia task definitions (parameterized)



Mapping specification

```
instance {
  name = matmul_node_inst
  variant = inner
  runs_at = main_memory
  tunable P=256, Q=256, R=256
}

instance {
  name = matmul_L2_inst
  variant = inner
  runs_at = L2_cache
  tunable P=32, Q=32, R=32
}

instance {
  name = matmul_L1_inst
  variant = leaf
  runs_at = L1_cache
```

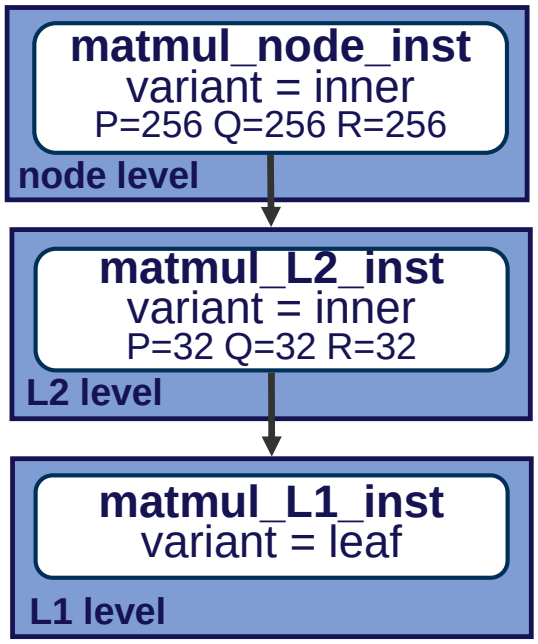
Task instances (not parameterized)



Task mapping specification

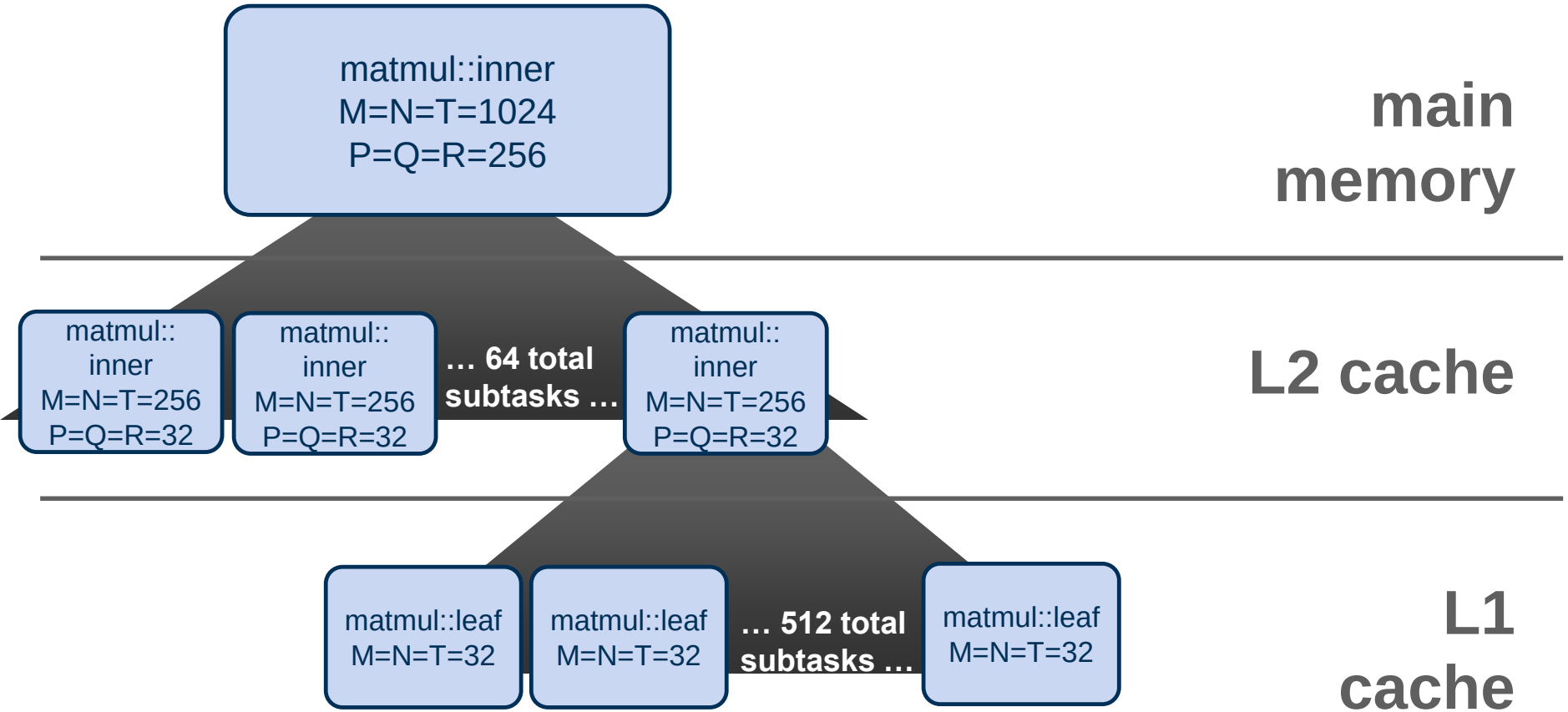
```
instance {
  name = matmul_node_inst
  task = matmul
  variant = inner
  runs_at = main_memory
  tunable P=256, Q=256, R=256
  calls = matmul_L2_inst
}

instance {
  name = matmul_L2_inst
  task = matmul
  variant = inner
  runs_at = L2_cache
  tunable P=32, Q=32, R=32
  calls = matmul_L1_inst
}

instance {
  name = matmul_L1_inst
  task = matmul
  variant = leaf
  runs_at = L1_cache
}
```

PC task instances



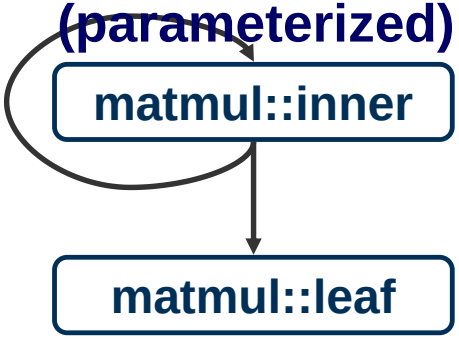
Specializing matmul

- Instances of tasks placed at each memory level



Task instances: Cell

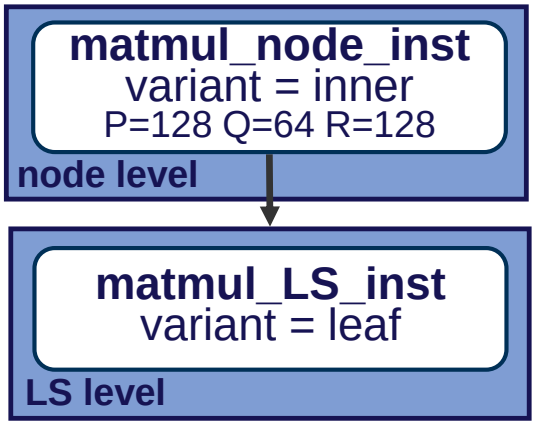
Sequoia task definitions (parameterized)



Cell mapping specification

```
instance {
  name = matmul_node_inst
  variant = inner
  runs_at = main_memory
  tunable P=128, Q=64, R=128
}

instance {
  name = matmul_LS_inst
  variant = leaf
  runs_at = LS_cache
}
```

Cell task instances (not parameterized)



Autotuning Sequoia Programs

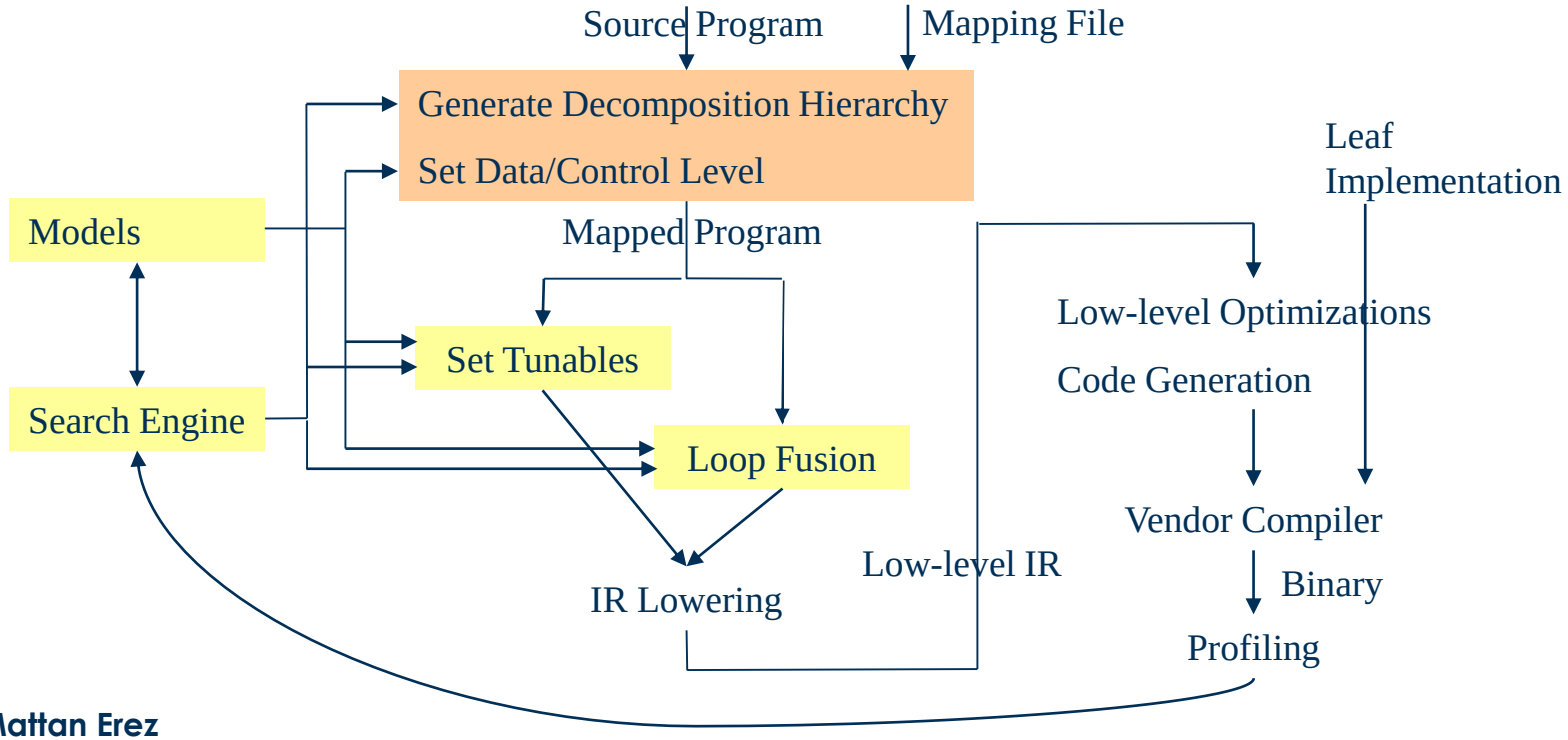
- Autotuner helps user with mapping (user can always override)

Specialization with Autotuning

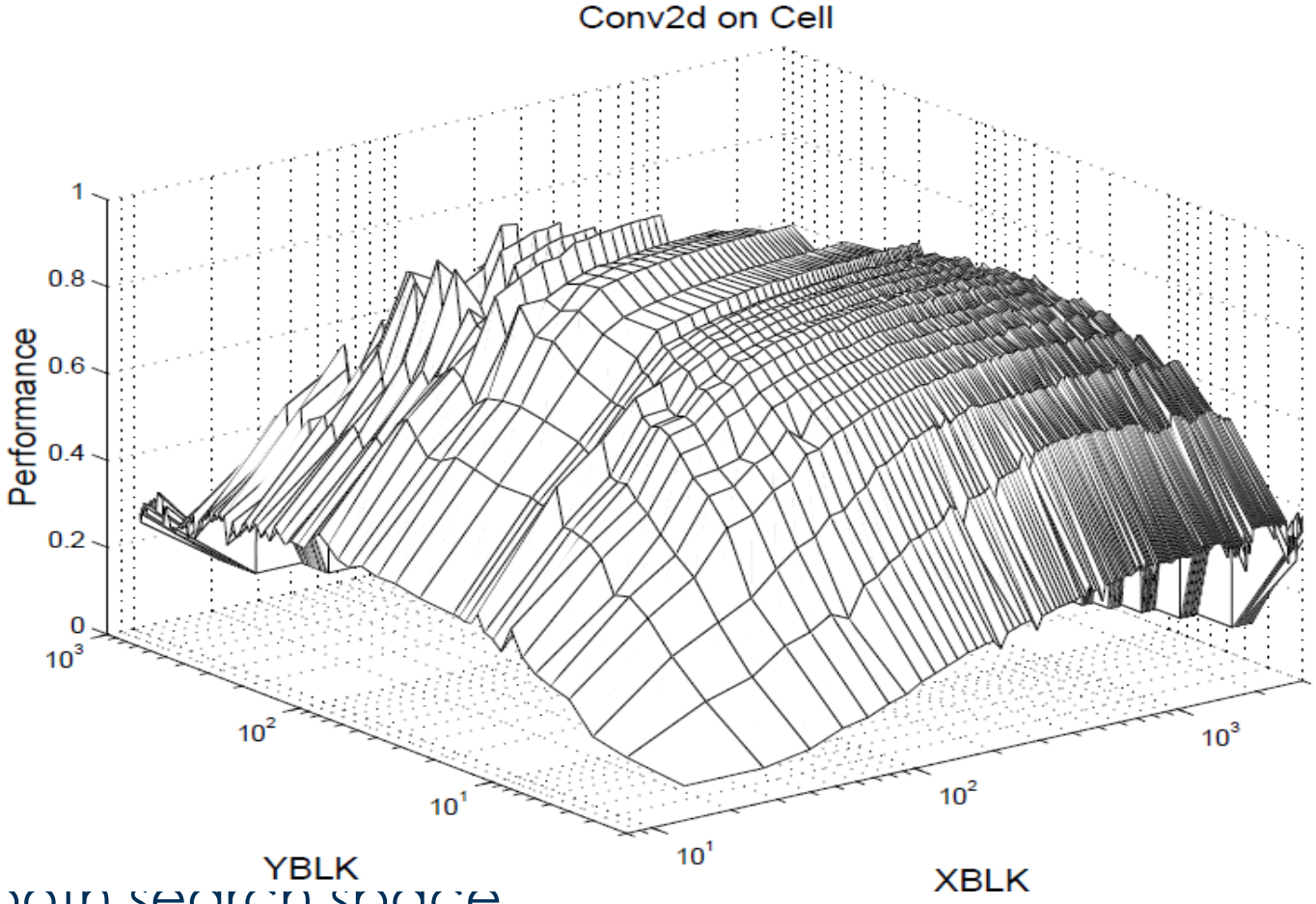
- Work by Manman Ren (Stanford), PACT 2008
- Use Sequoia to identify what needs tuning
  - Explicit tunables and parameters in the language
- Tuning framework for SW-managed hierarchies
- Automatic profile guided search across tunables
  - Aggressive pruning
  - Illegal parameters (don't fit in memory level)
  - Tunable groups
  - Programmer input on ranges
  - Coarse → fine search
- Loop fusion across multiple loop levels
  - Measure profitability from tunable search
  - Adjust for "tunable mismatch"
  - Realign reuse to reduce communication

Overview: mapping the program

- Mapped versions are generated
  - Matching the decomposition hierarchy with the machine hierarchy
  - Choosing a variant for each call site
  - Set level of data objects and control statements



Explicit SW Management Simplifies Tuning



- Smooth search space
- Performance models can also work
  - For Cell, not cluster

Autotuning OutPerforms Programmer

THE UNIVERSITY OF TEXAS AT AUSTIN

		CONV2D	SGEMM	FFT3D	SUmb
Cell	**auto** hand	**99.6** 85	**137** 119	**57** 54	**12.1**
Cluster of PCs	**auto** hand	**26.7** 24	**92.4** 90	**5.5** 5.5	**2.2**
Cluster of PS3s	**auto** hand	**20.7** 19	**33.4** 30	**0.57** 0.36	**0.63**

Summary of results

- Performance competitive with native code
  - Generic optimization for hierarchical bulk programs
    - Copy elimination
    - DMA transfer coalescing
    - Operation hoisting
    - Array allocation / packing
    - Scheduling (tasks and DMAs)
  - Automatic tuning for performance
- Portable: no source-code changes for different configurations
  - Cell, SMP, Cluster, Disk
  - Compositions of above
  - Automatic tuning
- Maximizes resources (compute or communication)
- Low overhead

Results

- We have a Sequoia compiler + runtime systems for multiple platforms
  - Cell/PS3
  - Cluster
  - Disk
  - SMP
- **Static compiler optimizations (bulk operation IR)**
  - **Copy elimination**
  - **DMA transfer coalescing**
  - **Operation hoisting**
  - **Array allocation / packing**
  - **Scheduling (tasks and DMAs)**
- **Runtimes can be composed**
  - **Cluster of PS3s**
  - **Disk + Cell**
  - **Cluster of SMPs**

Scientific computing benchmarks

Linear Algebra	Blas Level 1 SAXPY, Level 2 SGEMV, and Level 3 SGEMM benchmarks
Conv2D	2D convolution with 9x9 support (non-periodic boundary constraints)
FFT3D	256^3 complex FFT
Gravity	100 time steps of N-body stellar dynamics simulation
HMMER	Fuzzy protein string matching using HMM evaluation (Daniel Horn's SC2005 paper)

System configurations

- Disk
  - 2.4 GHz Intel P4, 160GB disk, ~50MB/s from disk
- 8-way SMP
  - 4 dual-core 2.66 Intel P4 Xeons, 8GB
- Cluster
  - 16, 2-way Intel 2.4GHz P4 Xeons, 1GB/node, Infiniband
- Cell
  - 3.2 GHz IBM Cell blade (8SPE), 1GB
- PS3
  - 3.2 GHz Cell in Sony Playstation 3 (6 SPE), 256MB (160MB usable)

Results – Horizontal portability - GFlop/s

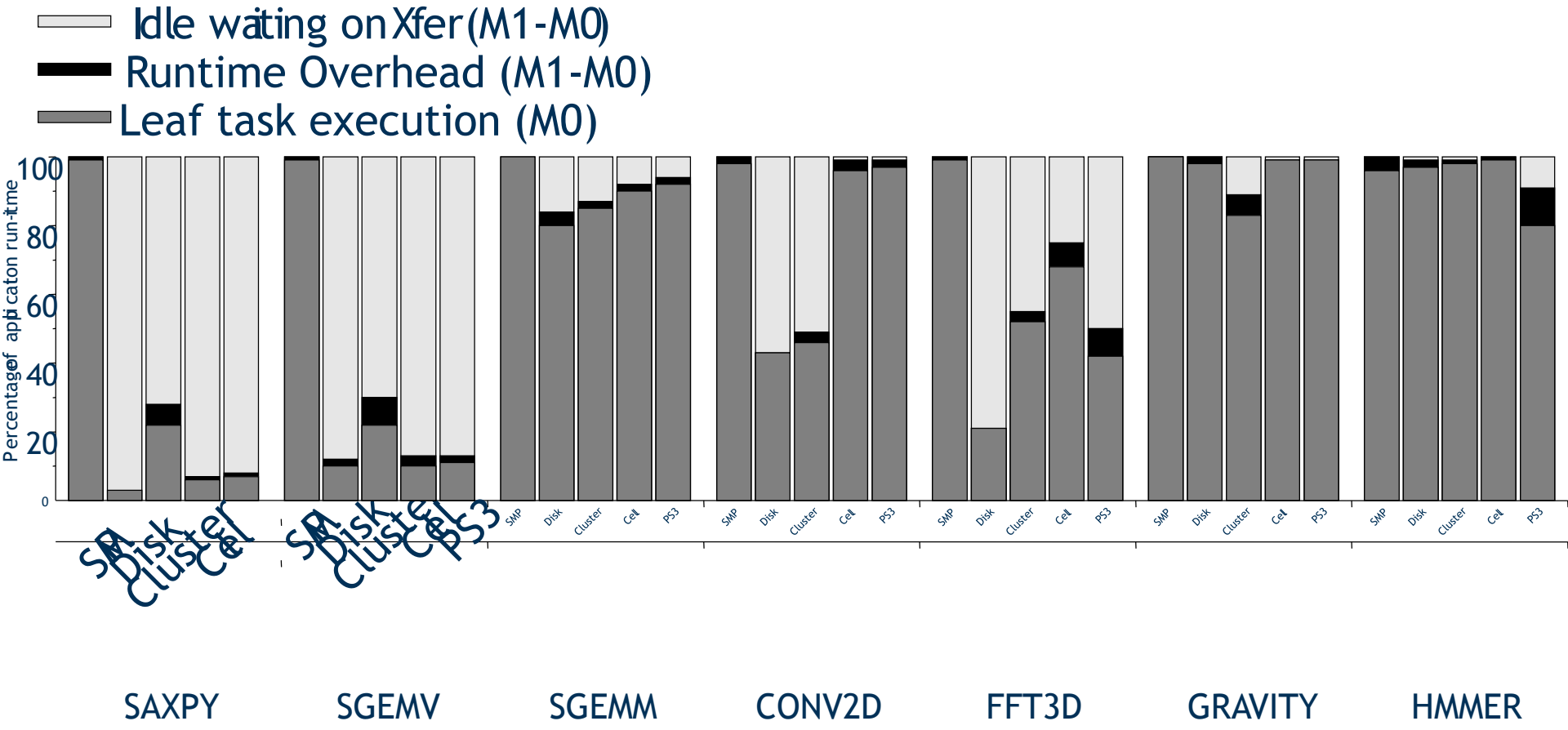
	Scalar	SMP	Disk	Cluster	Cell	PS3
SAXPY	0.3	0.7	0.007	1.4	3.5	3.1
SGEMV	1.1	1.7	0.04	3.8	12	10
SGEMM	6.9	45	5.5	91	119	94
CONV2D	1.9	7.8	0.6	24	85	62
FFT3D	1.5	7.8	0.1	7.5	54	31*
GRAVITY	4.8	40	3.7	68	97	71
HMMER	0.9	11	0.9	12	12	7.1*

Results – Horizontal portability - GFlop/s

	Scalar	SMP	Disk	Cluster	Cell	PS3
SAXPY	0.3	0.7	0.007	1.4	3.5	3.1
SGEMV	1.1	1.7	0.04	3.8	12	10
SGEMM	6.9	45	5.5	91	119	94
CONV2D	1.9	7.8	0.6	24	85	62
FFT3D	1.5	7.8	0.1	7.5	54	31*
GRAVITY	4.8	40	3.7	68	97	71
HMMER	0.9	11	0.9	12	12	7.1*

Bandwidth bound

2 Level Utilization



Results – Vertical Portability - GFlop/s

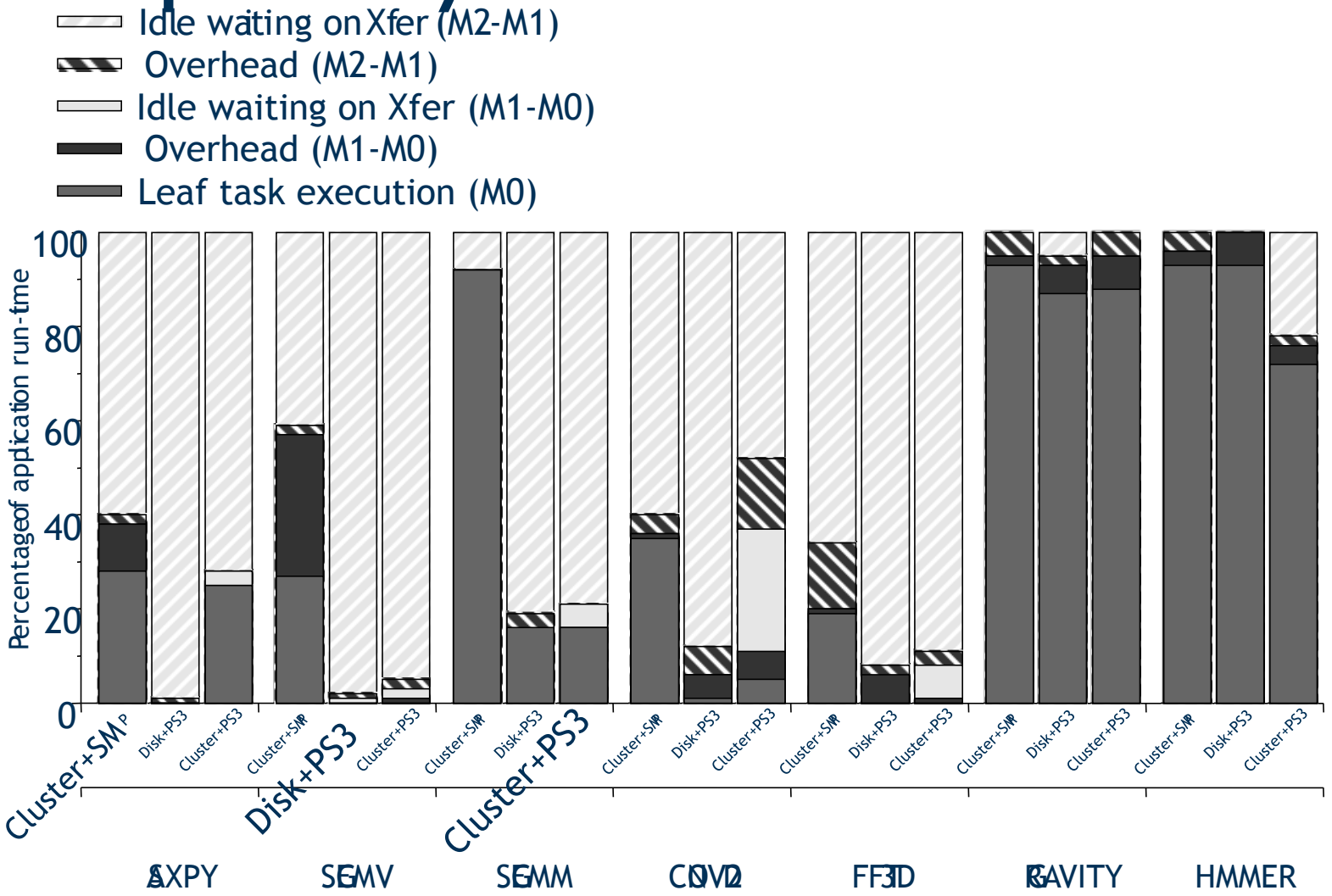
	Cluster-SMP	Disk+PS3	PS3 Cluster
SAXPY	0.5	0.004	0.23
SGEMV	1.4	0.014	1.3
SGEMM	48	3.7	30
CONV2D	4.8	0.48	3.24
FFT3D	2.1	0.05	0.36
GRAVITY	50	66	119
HMMER	14	8.3	13

Results – Vertical Portability - GFlop/s

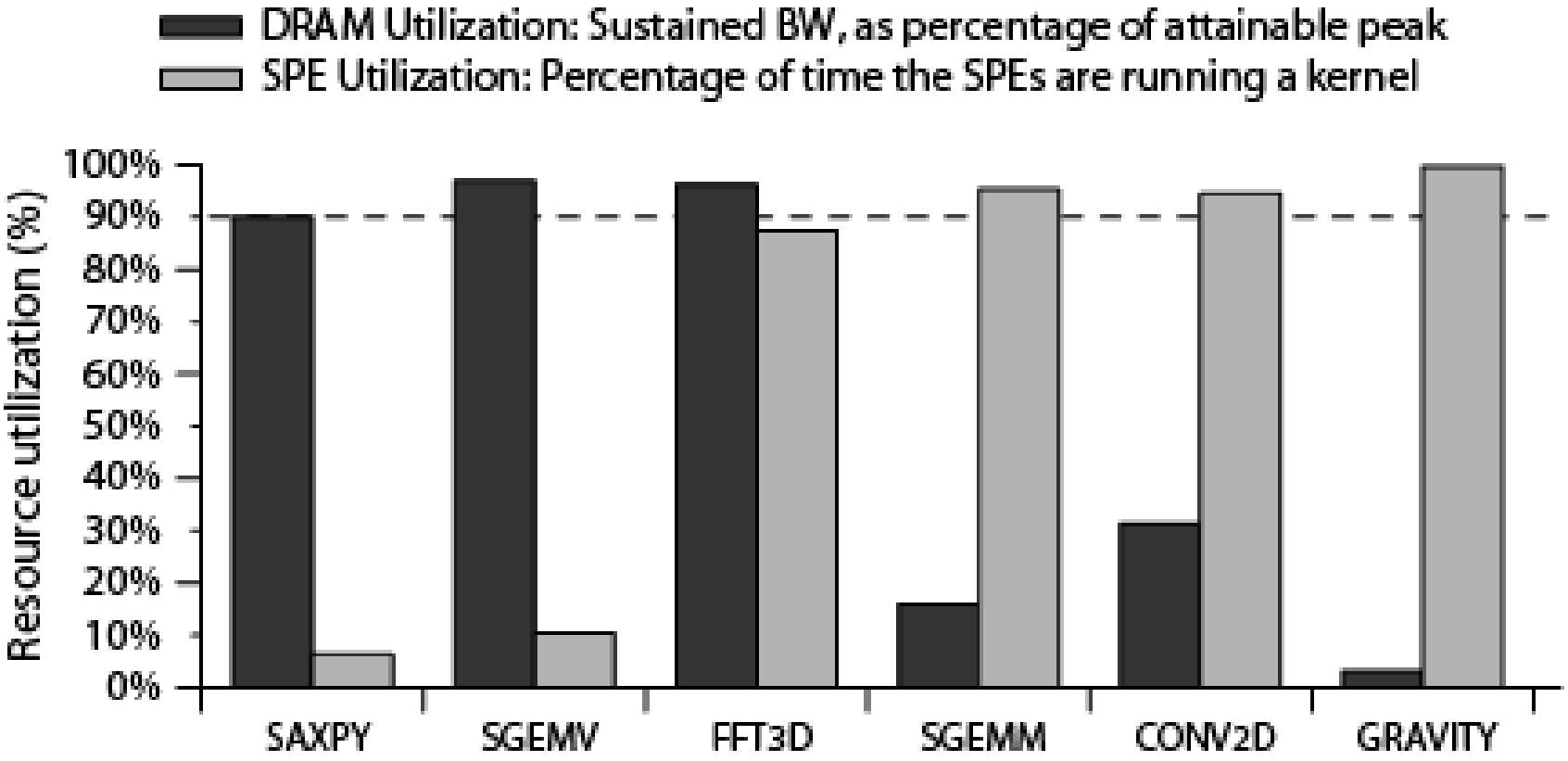
	Cluster-SMP	Disk+PS3	PS3 Cluster
SAXPY	0.5	0.004	0.23
SGEMV	1.4	0.014	1.3
SGEMM	48	3.7	30
CONV2D	4.8	0.48	3.24
FFT3D	2.1	0.05	0.36
GRAVITY	50	66	119
HMMER	14	8.3	13

Bandwidth bound

Composed systems utilization

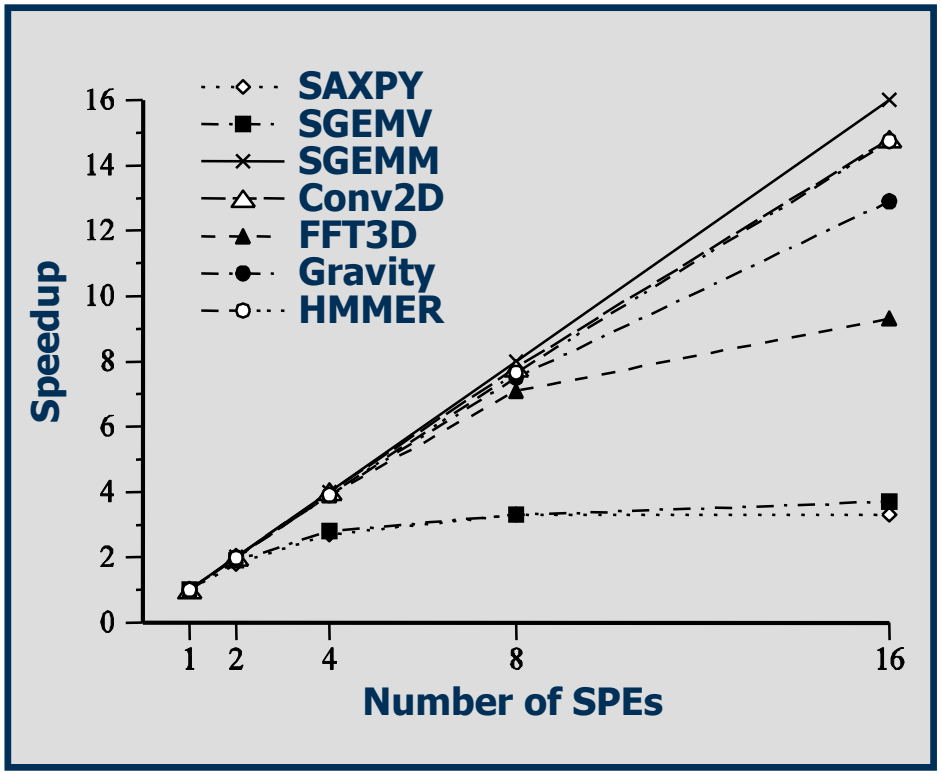


Cell utilization

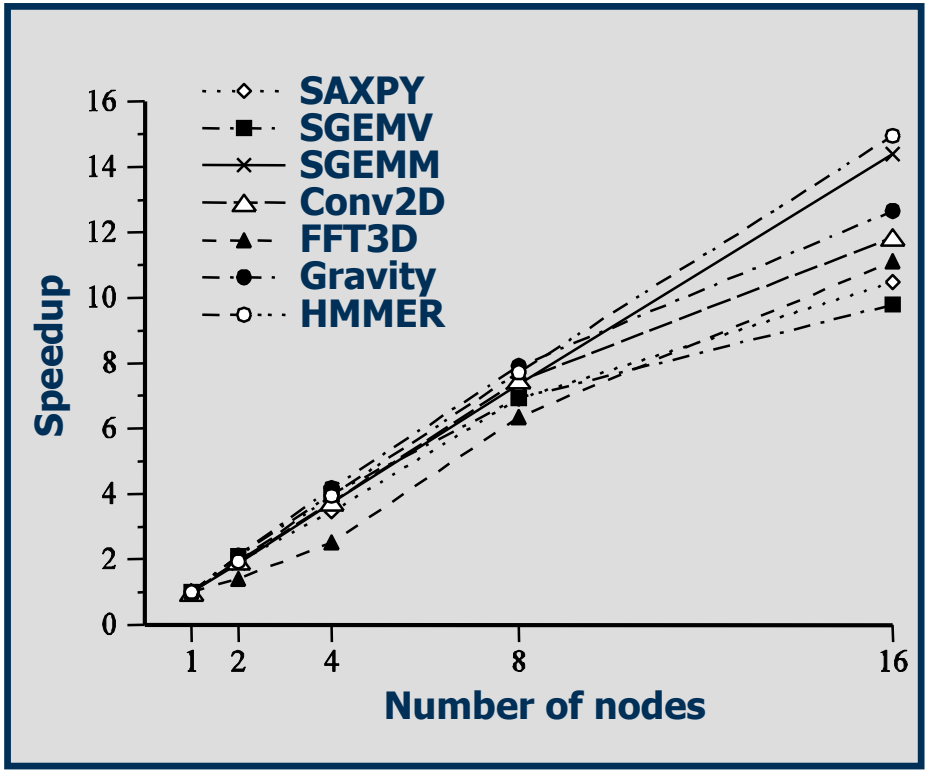


Performance scaling

SPE scaling on 2.4GHz Dual-Cell blade



Scaling on P4 cluster with Infiniband interconnect



Key ideas

- Incorporate hierarchal memory tightly into programming model
  - Programming memory hierarchy
- Abstract [horizontal + vertical] communication and locality
  - Vertical portability
- Leverage task abstraction for critical properties of application
- Mapping to specific machine is separate from algorithm description

Sequoia limitations

- Require explicit declaration of working sets
  - Programmer must know what to transfer
  - Some irregular applications present problems
- Task mapping somewhat laborious
  - Autotuning helps
  - **Understand which parts can be automated better**
- Leaf-tasks (kernels) are external
  - Limits potential optimizations
  - Cumbersome