Stream Programming: Explicit Parallelism and Locality

Bill Dally

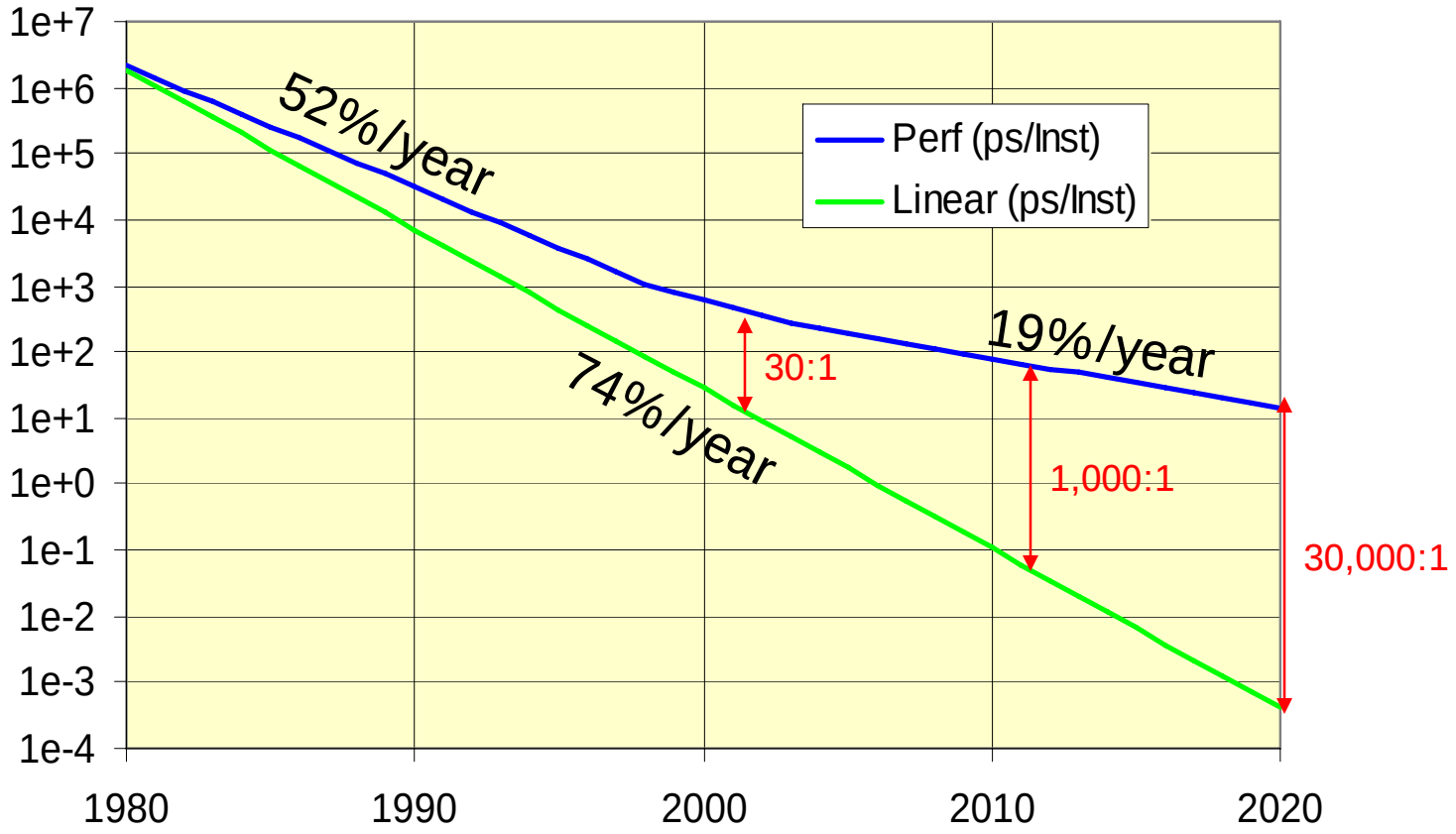
Edge Workshop

May 24, 2006

Outline

- Technology Constraints ➔ Architecture
- Stream programming
- Imagine and Merrimac
- Other stream processors
- Future directions

ILP is mined out – end of superscalar processors Time for a new architecture



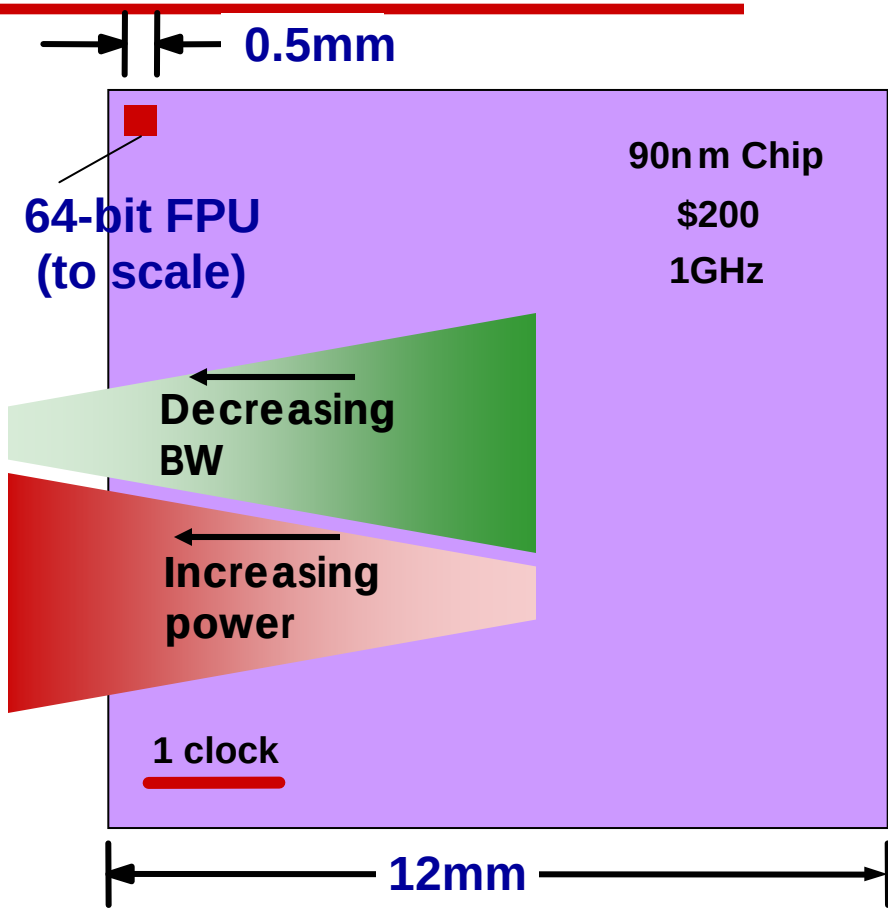
Dally et al. “The Last Classsical Computer”, ISAT Study, 2001

Performance = Parallelism

Efficiency = Locality

Arithmetic is cheap, Communication is expensive

- **Arithmetic**
  - Can put 100s of FPUs on a chip
  - $0.50/GFLOPS, 50mW/GFLOPS
  - Exploit with **parallelism**
- **Communication**
  - Dominates cost
    - $8/GW/s 2W/GW/s (off-chip)
  - BW decreases (and cost increases) with distance
  - Power increases with distance
  - Latency increases with distance
    - But can be hidden with parallelism
  - Need **locality** to conserve global bandwidth

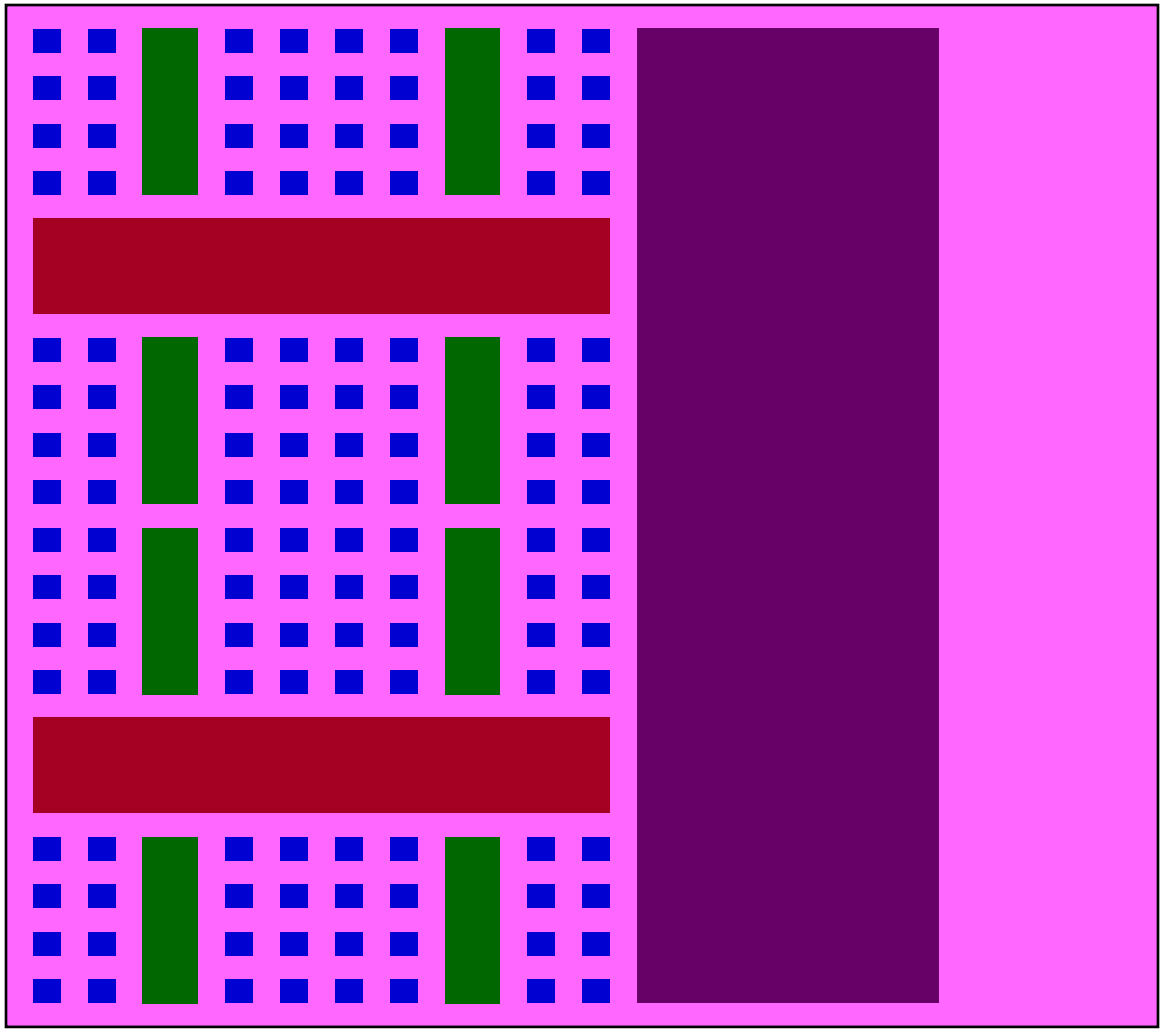


Cost of data access varies by 1000x

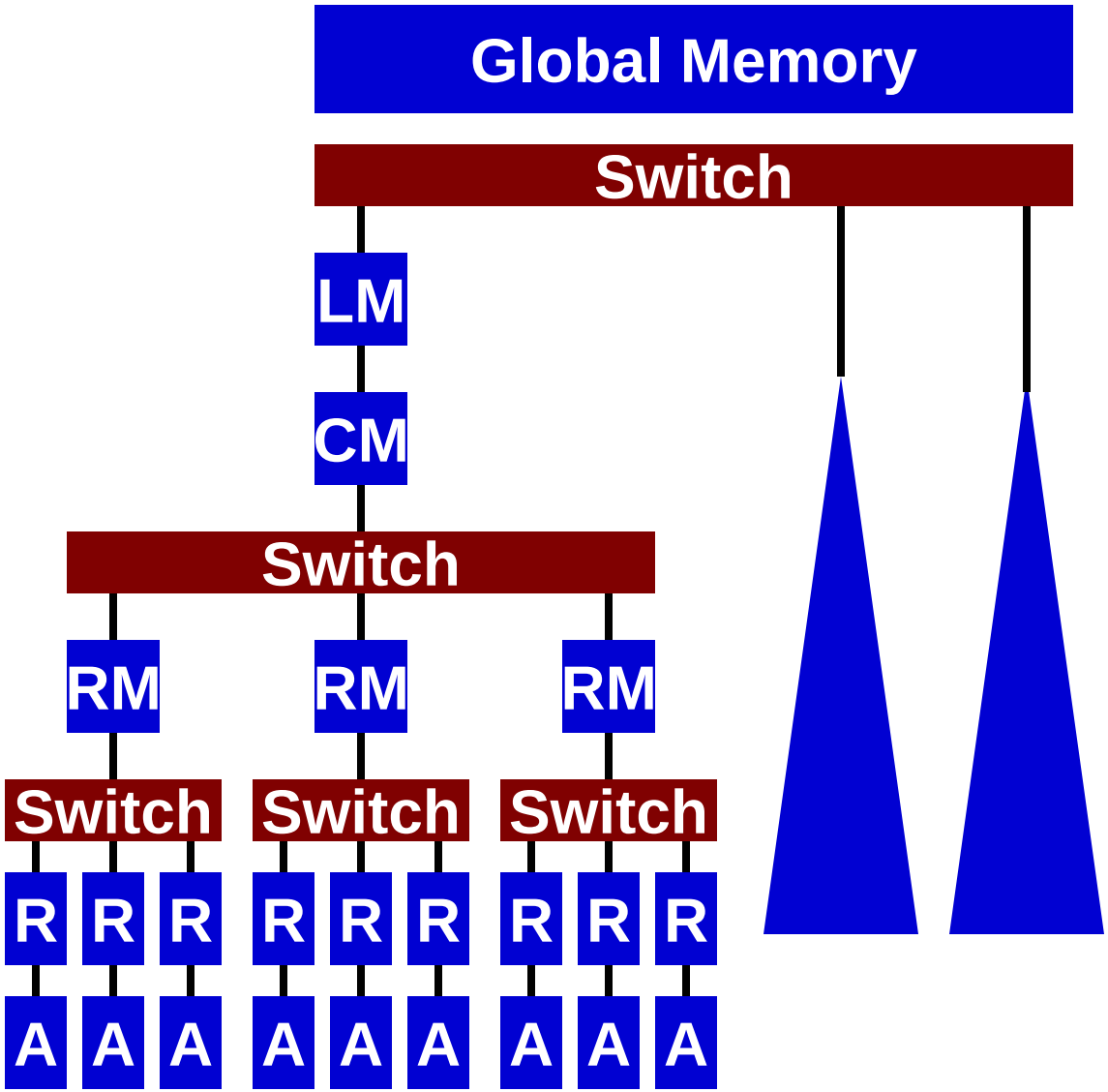
From	Energy	Cost*	Time
Local Register	10pJ	$0.50	1ns
Chip Region (2mm)	50pJ	$2	4ns
Global on Chip (15mm)	200pJ	$10	20ns
Off chip (node mem)	1nJ	$50	200ns
Global	5nJ	$500	1us

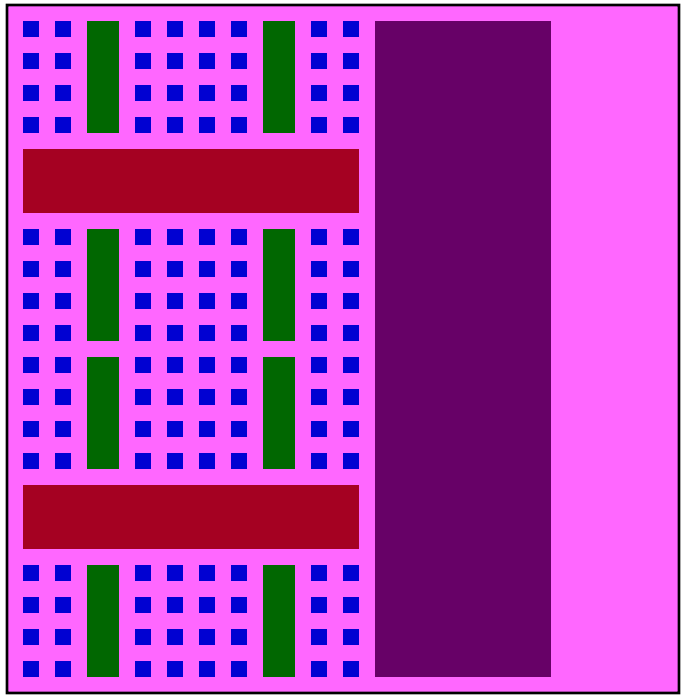
*Cost of providing 1GW/s of bandwidth
All numbers approximate

So we should build chips that look like this



An abstract view

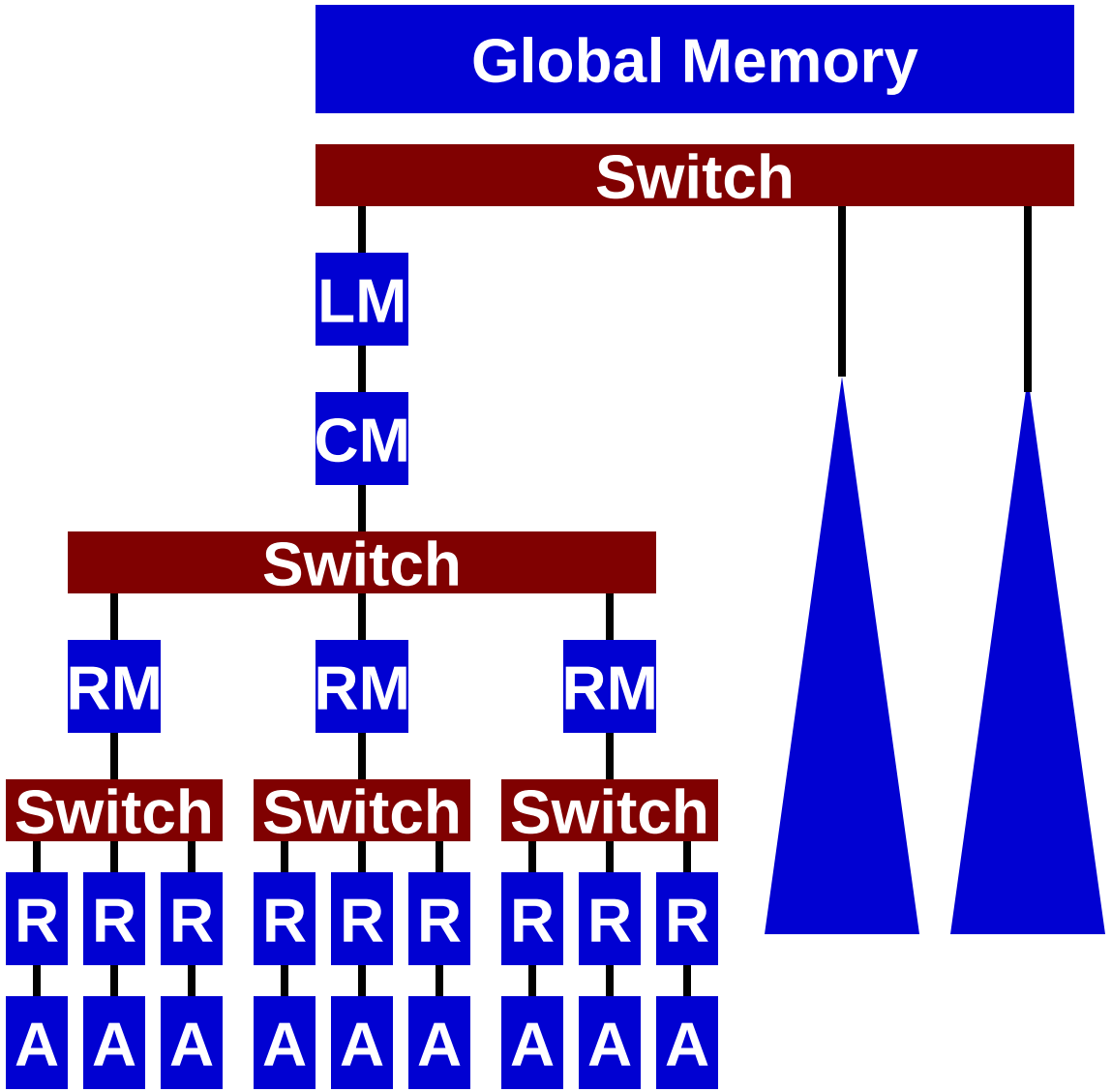




Real question is:

How to orchestrate movement of data

Conventional Wisdom: Use caches



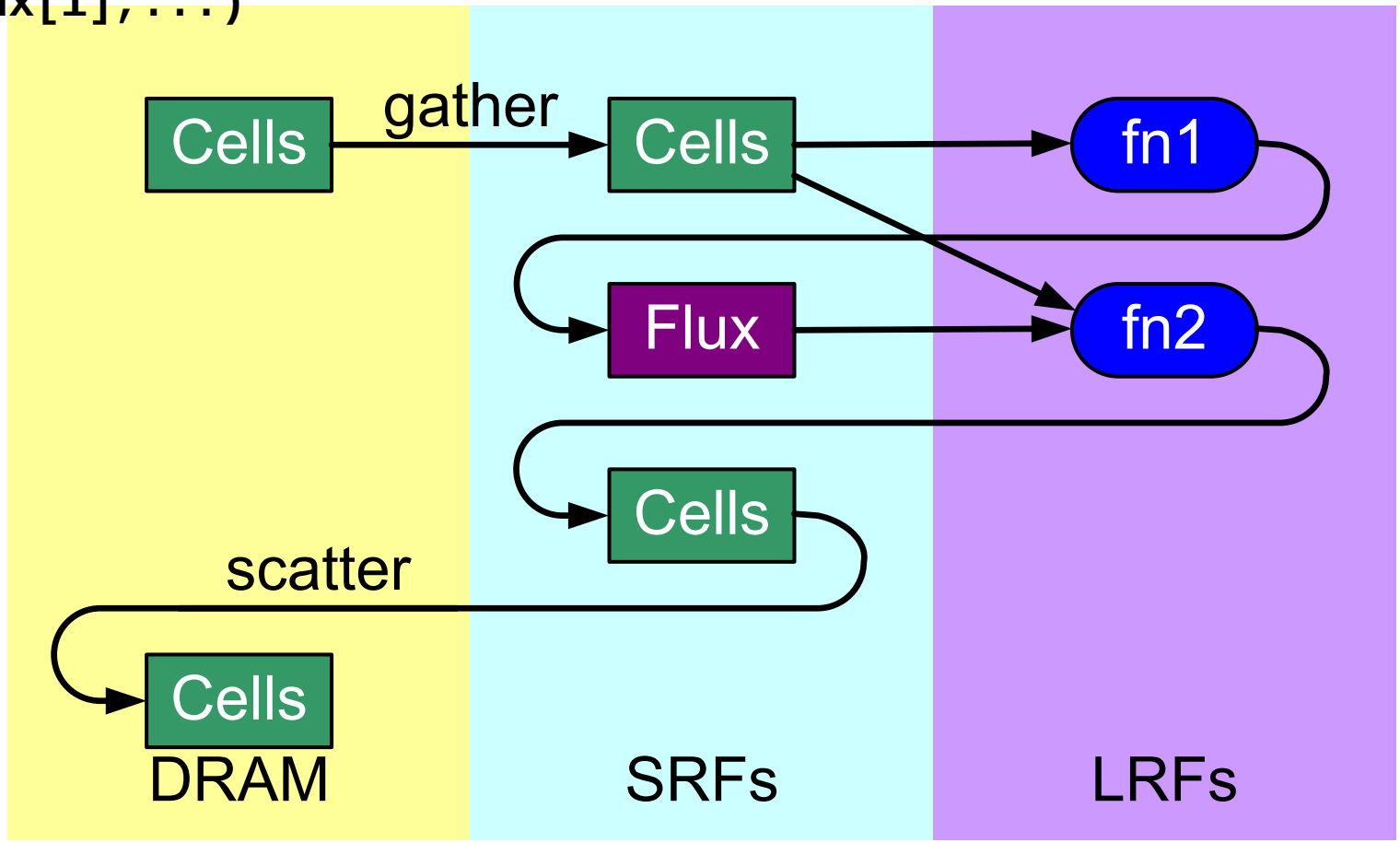
Caches squander bandwidth – our scarce resource

- Unnecessary data movement
- Poorly scheduled data movement
  - Idles expensive resources waiting on data

- More efficient to map programs to an explicit memory hierarchy

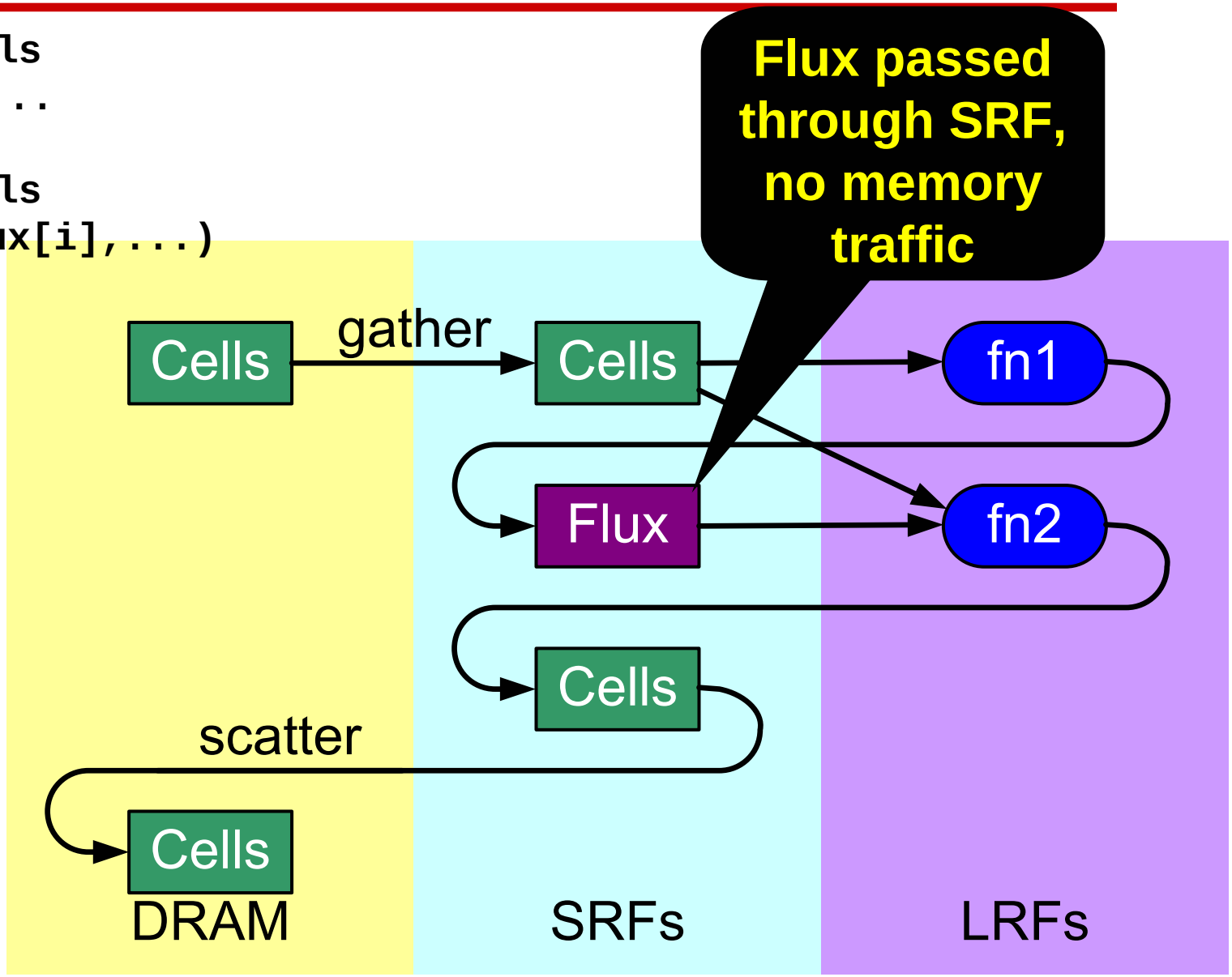
Example – Simplified Finite-Element Code

```
loop over cells
   flux[i] = ...

loop over cells
   ... = f(flux[i],...)
```



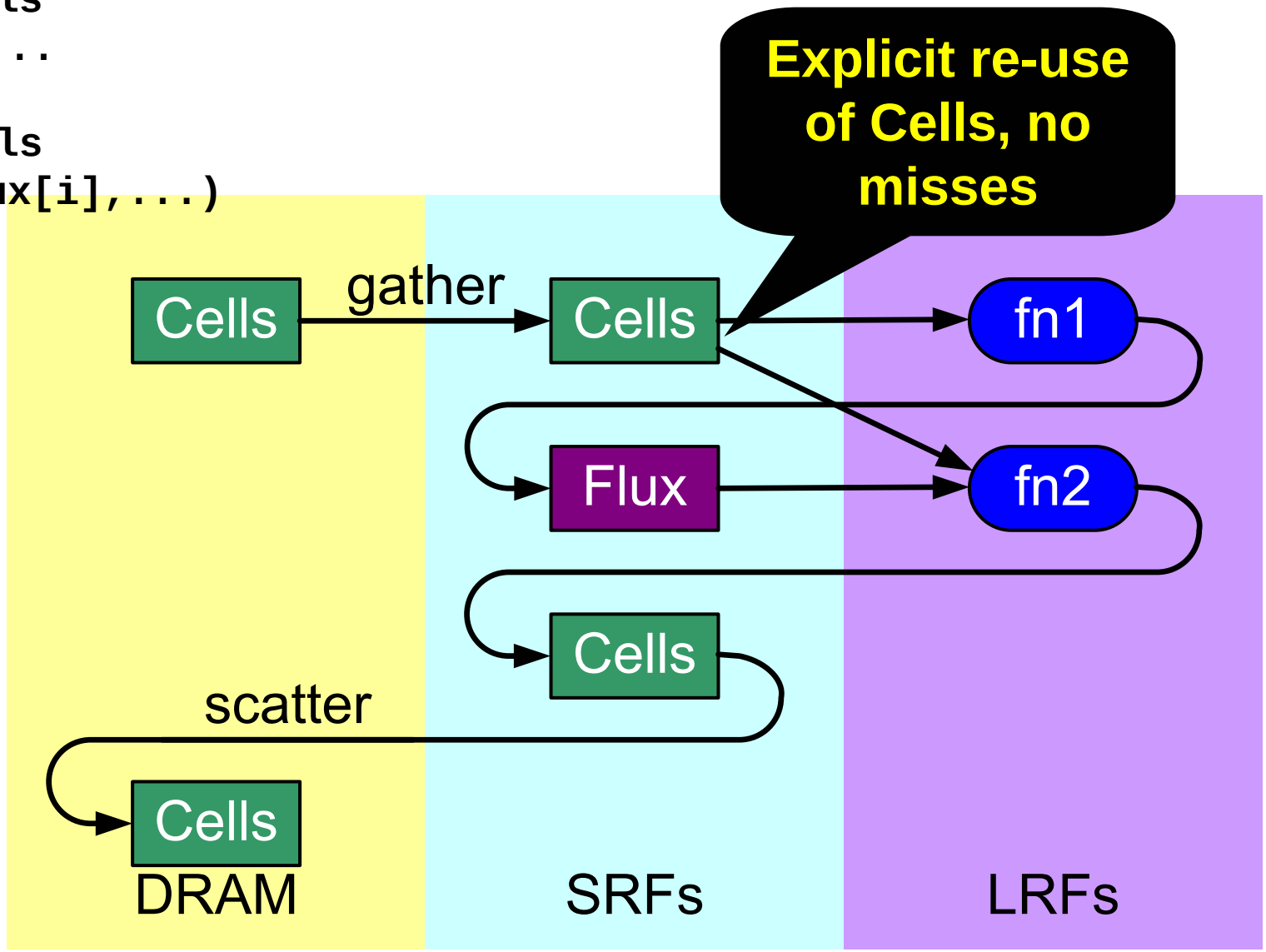
Explicitly block into SRF

```
loop over cells
   flux[i] = ...

loop over cells
   ... = f(flux[i],...)
```

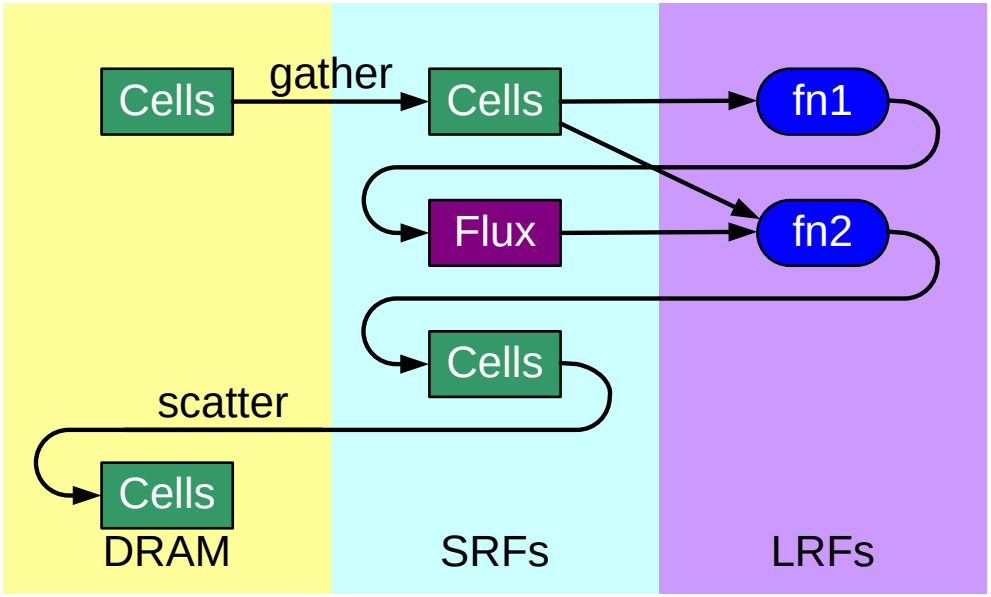


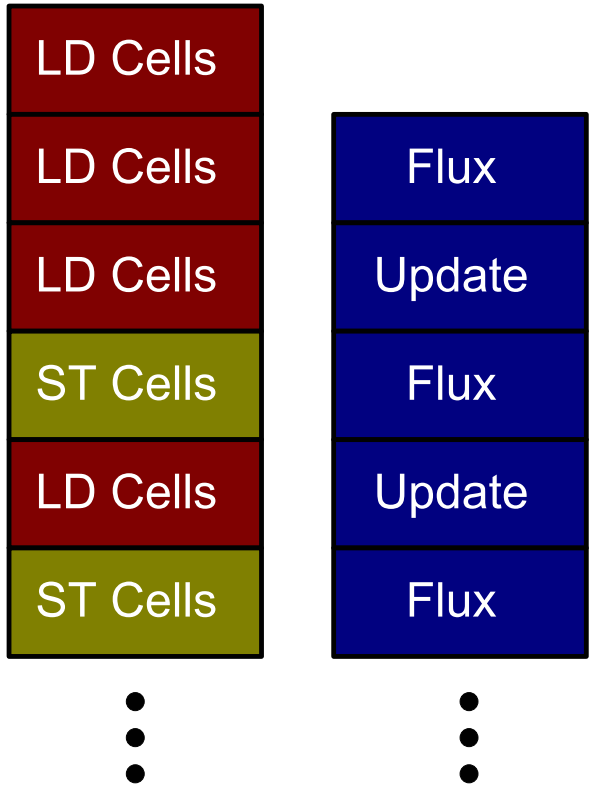
Explicitly block into SRF

```
loop over cells
   flux[i] = ...

loop over cells
   ... = f(flux[i],...)
```

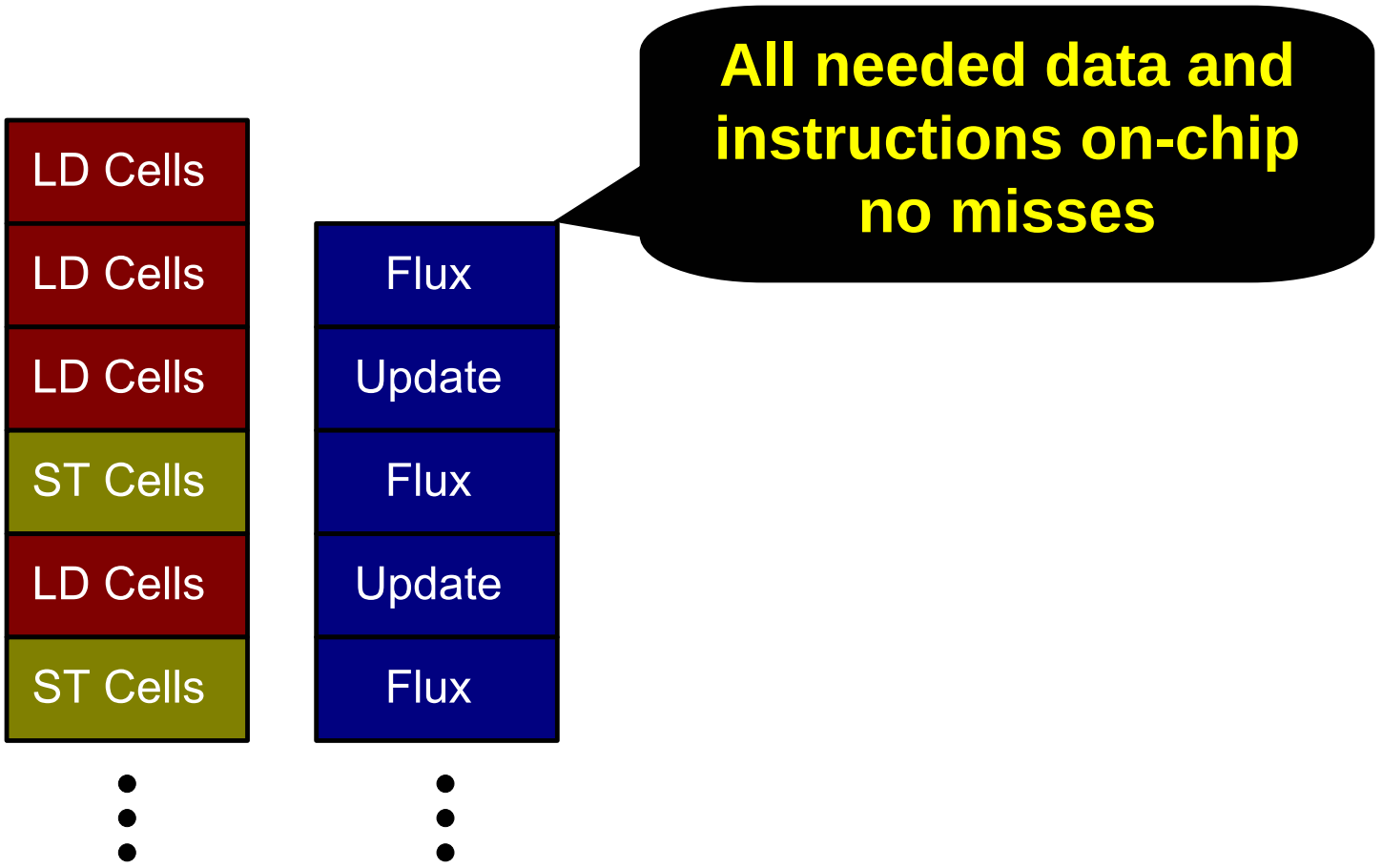


Stream loads/stores (bulk operations) hide latency (1000s of words in flight)

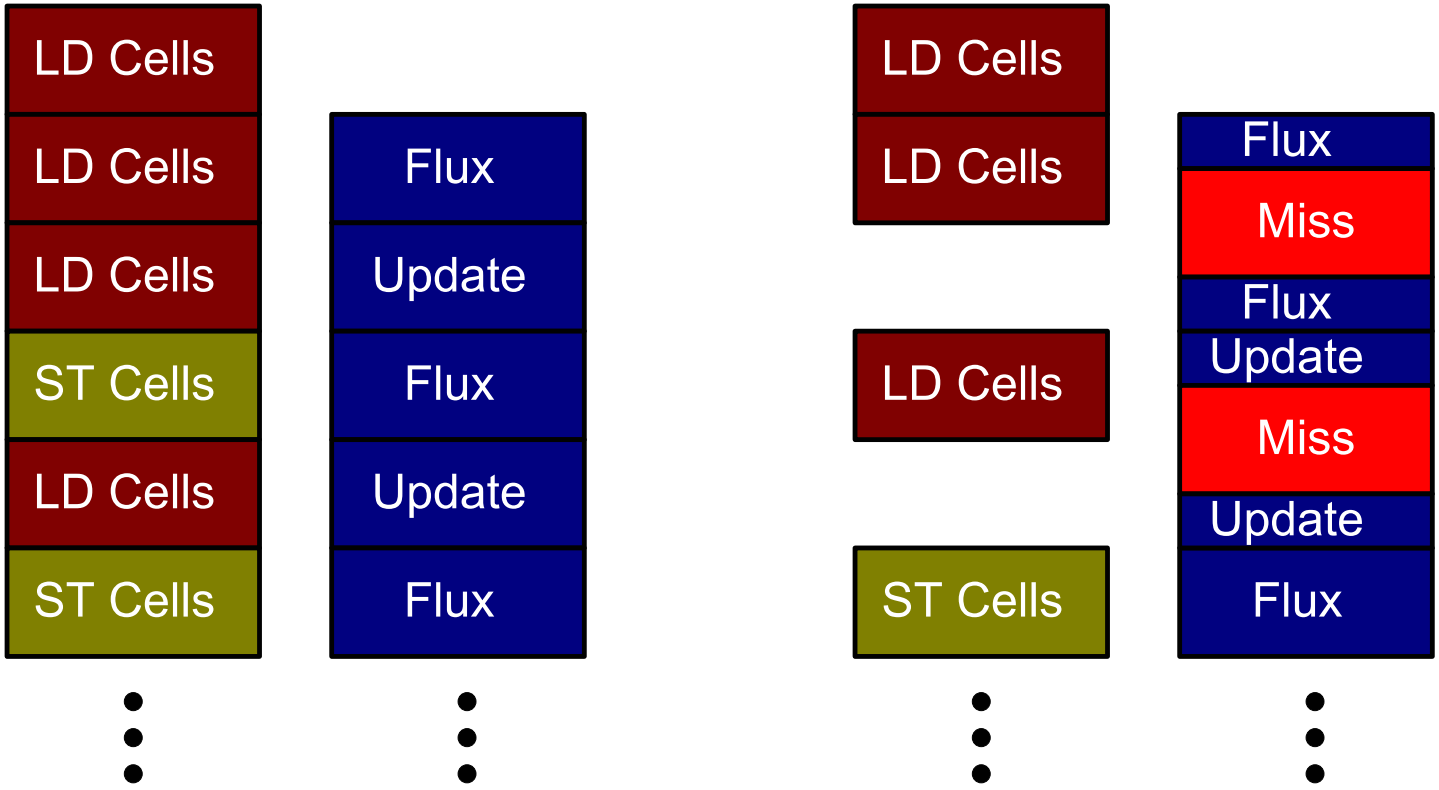




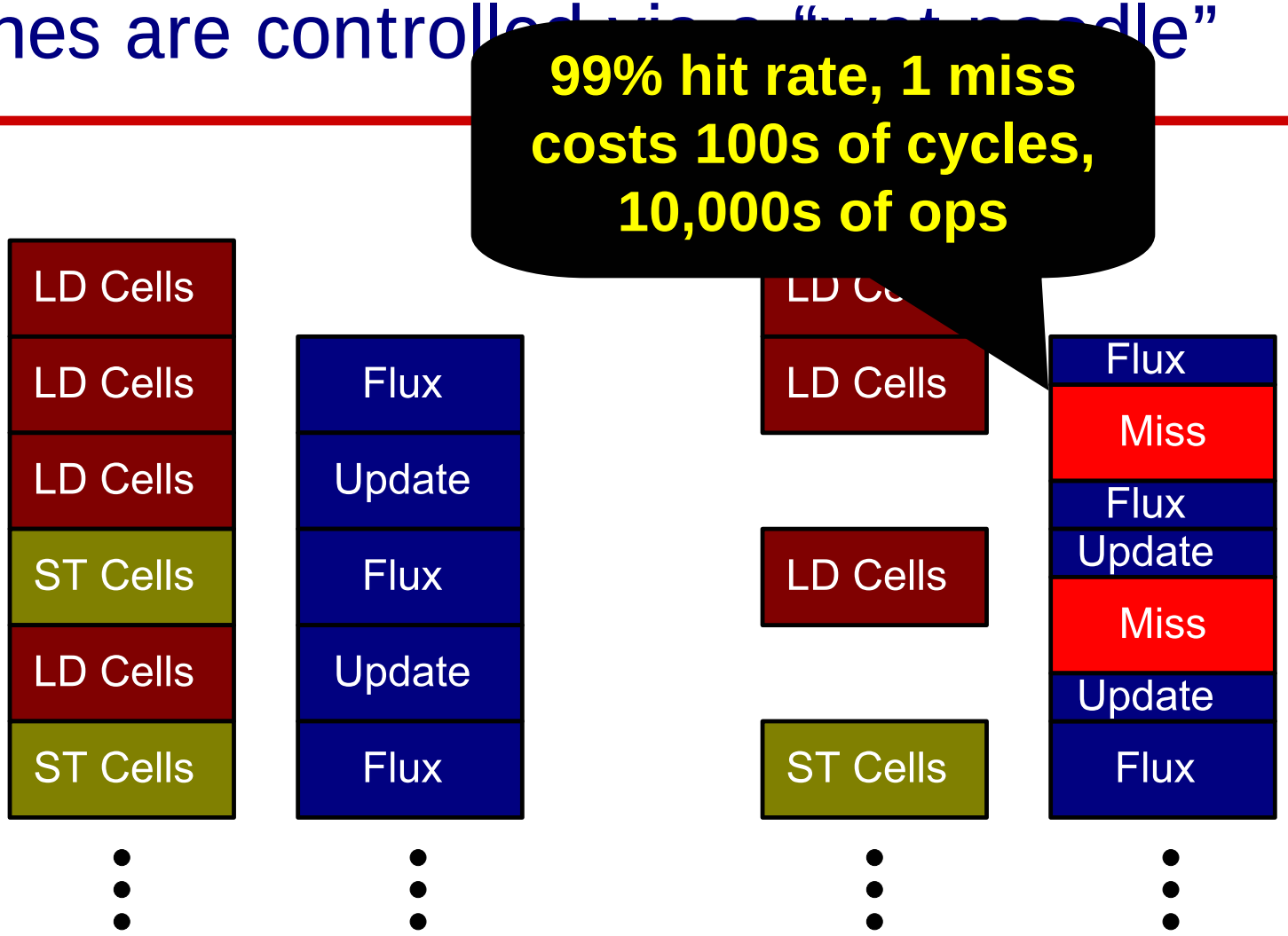
Explicit storage enables simple, efficient execution



Caches lack predictability (controlled via a “wet noodle”)



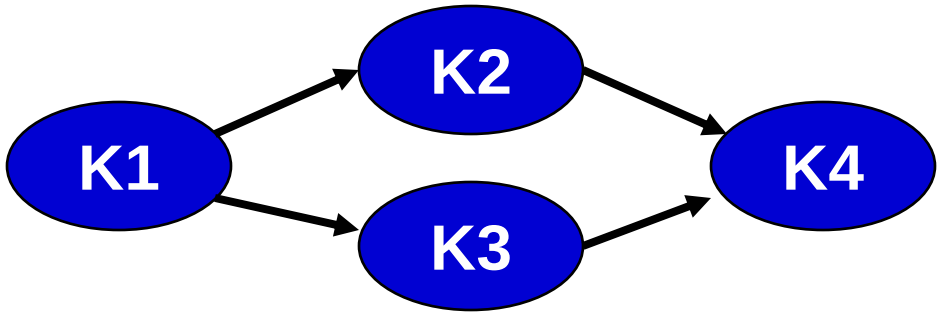
Caches are controlled via a "wet noodle"



So how do we program an explicit hierarchy?

Stream Programming: Parallelism, Locality, and Predictability

- Parallelism
  - Data parallelism across stream elements
  - Task parallelsm across kernels
  - ILP within kernels
- Locality
  - Producer/consumer
  - Within kernels
- Predictability
  - Enables scheduling



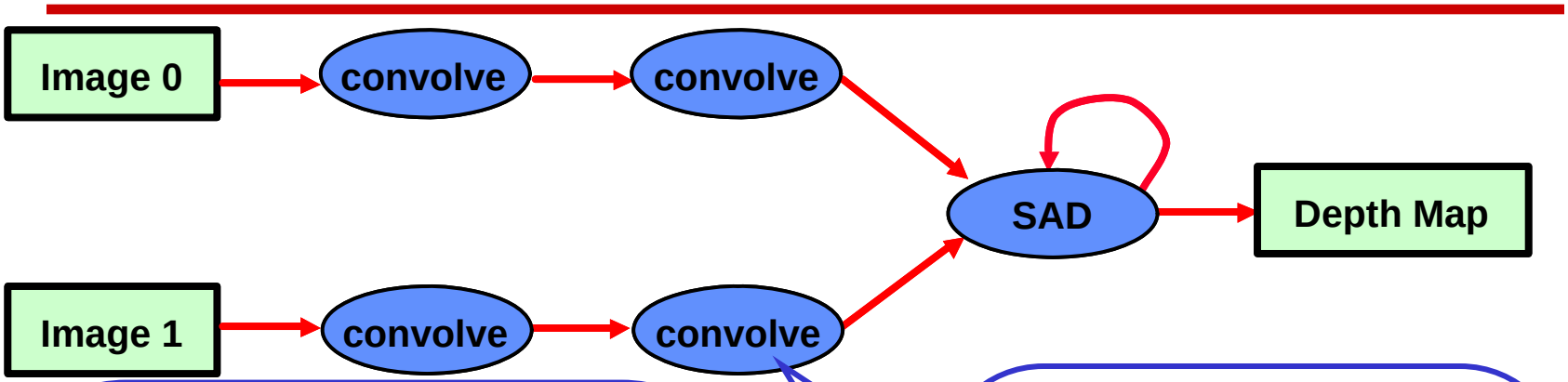
Evolution of Stream Programming

1997 StreamC/KernelC
Break programs into kernels
Kernels operate only on input/output streams and locals
Communication scheduling and stream scheduling

2001 Brook
Continues the construct of streams and kernels
Hides underlying details
Too “one-dimensional”

2005 Sequoia
Generalizes kernels to “tasks”
Tasks operate on local data
Local data “gathered” in an arbitrary way
“Inner” tasks subdivide, “leaf” tasks compute
Machine-specific details factored out

StreamC/KernelC

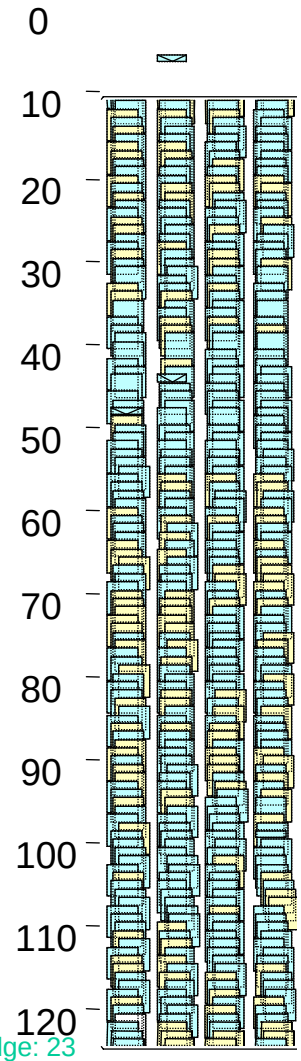



```
STREAMPROG depth) {
  im_stream<pixels> in, tmp;
  …
  for (i=0; i<rows; i++) {
    convolve(in, tmp, …);
    convolve(tmp, conv_row, …);
  }
  …
  for (i=0; i<rows; i++) {
    SAD(conv_row, depth_row, …);
  }
  …
}
```

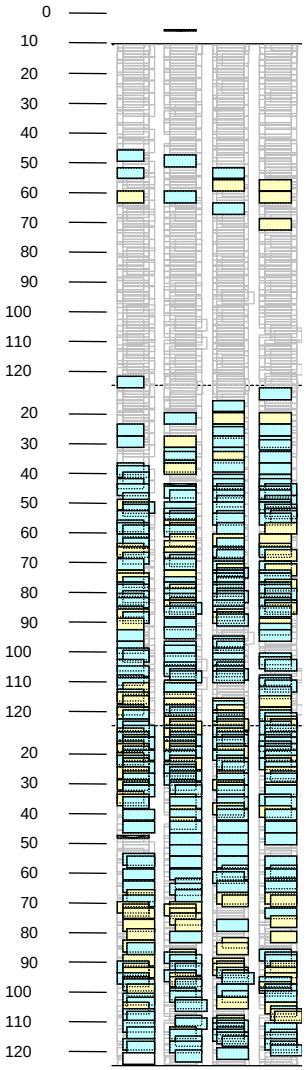
```
KERNEL convolve(
  istream<int> a,
  ostream<int> y) {
  …
  loop_stream(a) {
    int ai, out;
    a >> ai;
    …
    out = dotproduct(ai,…);
    y << out;
  }
}
```

Explicit storage enables simple, efficient execution unit scheduling

One iteration



SW Pipeline



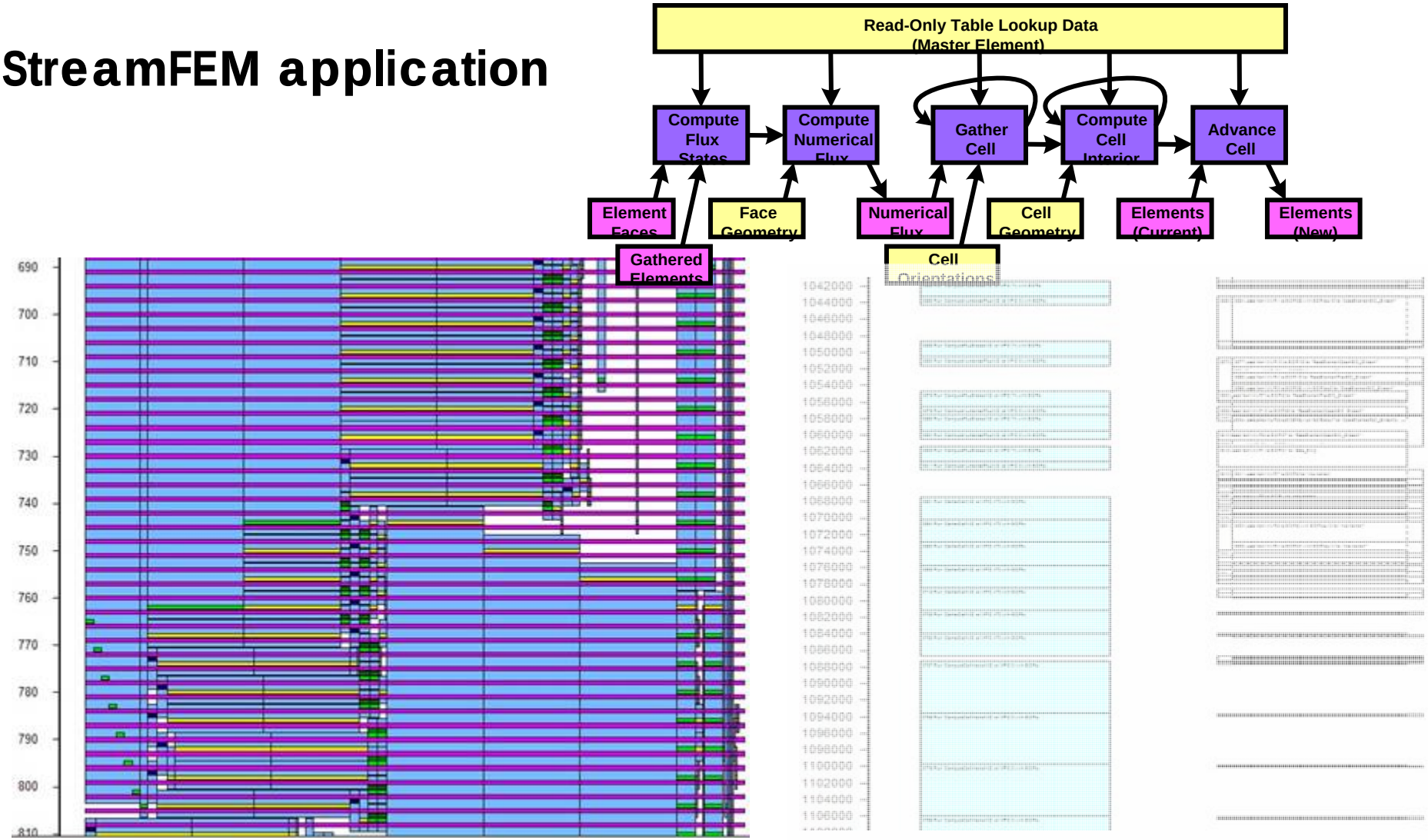
ComputeCellInt kernel from StreamFem3D

Over 95% of peak with simple hardware

Depends on explicit communication to make delays predictable

Stream scheduling exploits explicit storage to reduce bandwidth demand

StreamFEM application

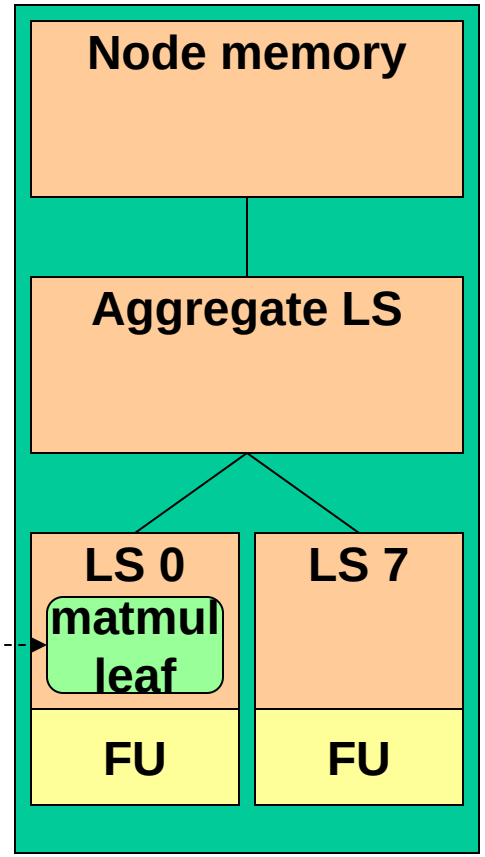


Prefetching, reuse, use/def, limited spilling

Sequoia – Generalize Kernels into Leaf Tasks

- Perform actual computation
- Analogous to kernels
- “Small” working set

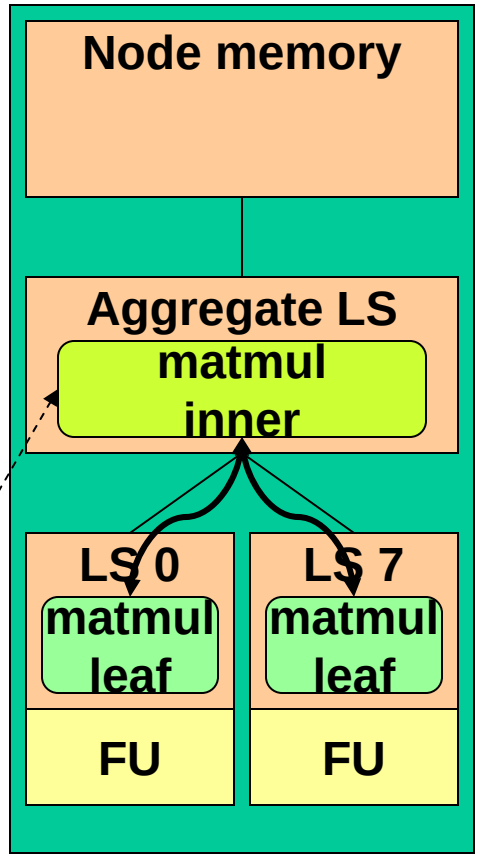
```
void __task matmul::leaf( __in    float A[M][P],
                          __in    float B[P][N],
                          __inout float C[M][N] )
{
  for (int i=0; i<M; i++) {
    for (int j=0; j<N; j++) {
      for (int k=0; k<P; k++) {
        C[i][j] += A[i][k] * B[k][j];
}
```



Inner tasks

- Decompose to smaller subtasks
  - Recursively
- “Larger” working sets

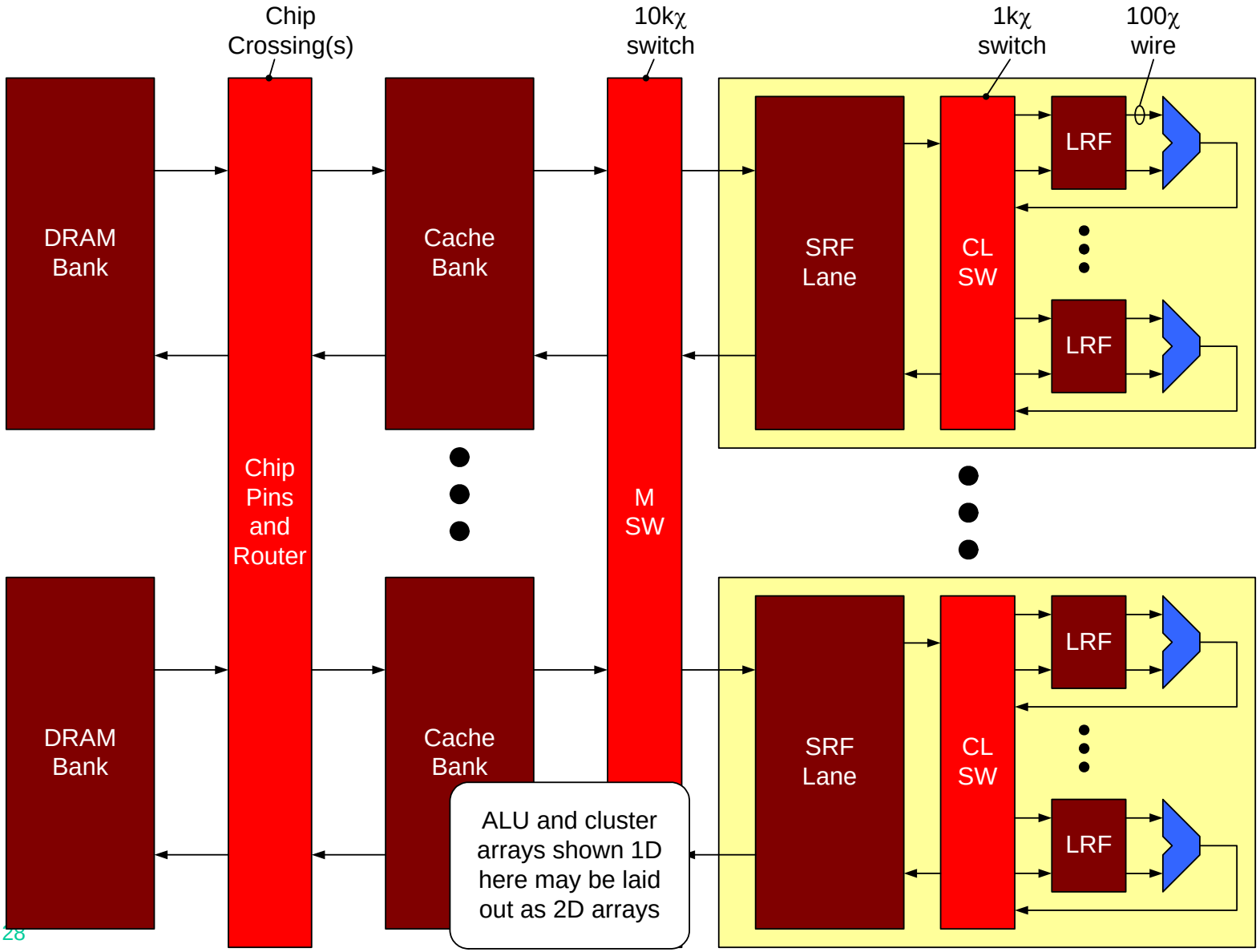
```
void __task matmul::inner( __in    float A[M][P],
                           __in    float B[P][N],
                           __inout float C[M][N] )
{
  tunable unsigned int U, X, V;
  blkset Ablks = rchop(A, U, X);
  blkset Bblks = rchop(B, X, V);
  blkset Cblks = rchop(C, U, V);

  mappar (int i=0 to M/U, int j=0 to N/V)
    mapreduce (int k=0 to P/X)

      matmul(Ablks[i][k],Bblks[k][j],Cblks[i][j]);
}
```



Stream Processors make communication explicit

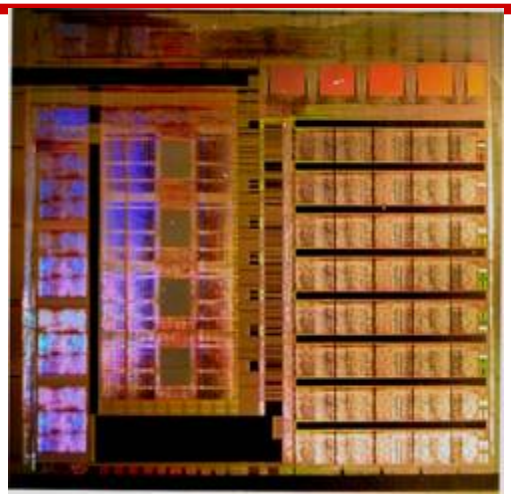
Enables optimization

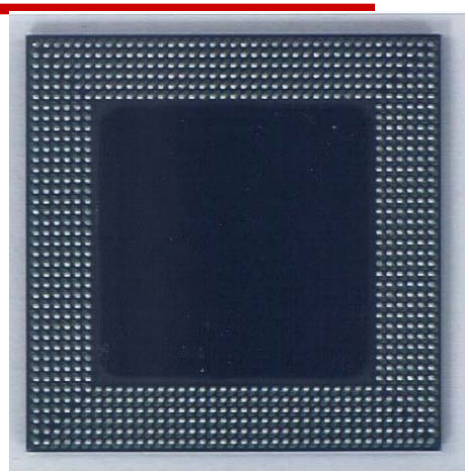
Stream architecture makes communication explicit – exploits parallelism and locality



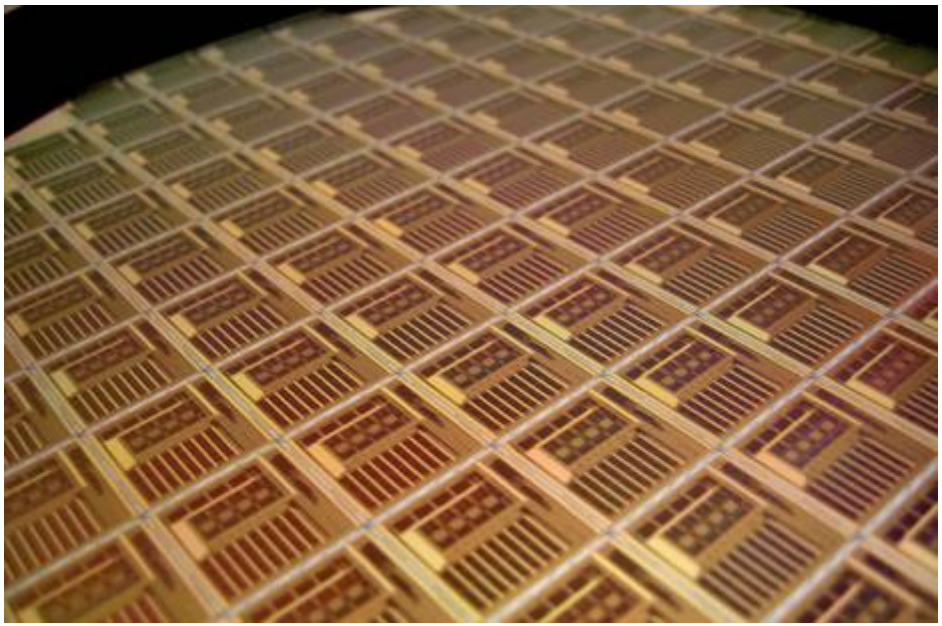
Imagine VLSI Implementation

- Chip Details
  - 2.56cm^2 die, 0.15um process, 21M transistors, 792-pin BGA
  - Collaboration with TI ASIC
  - Chips arrived on April 1, 2002

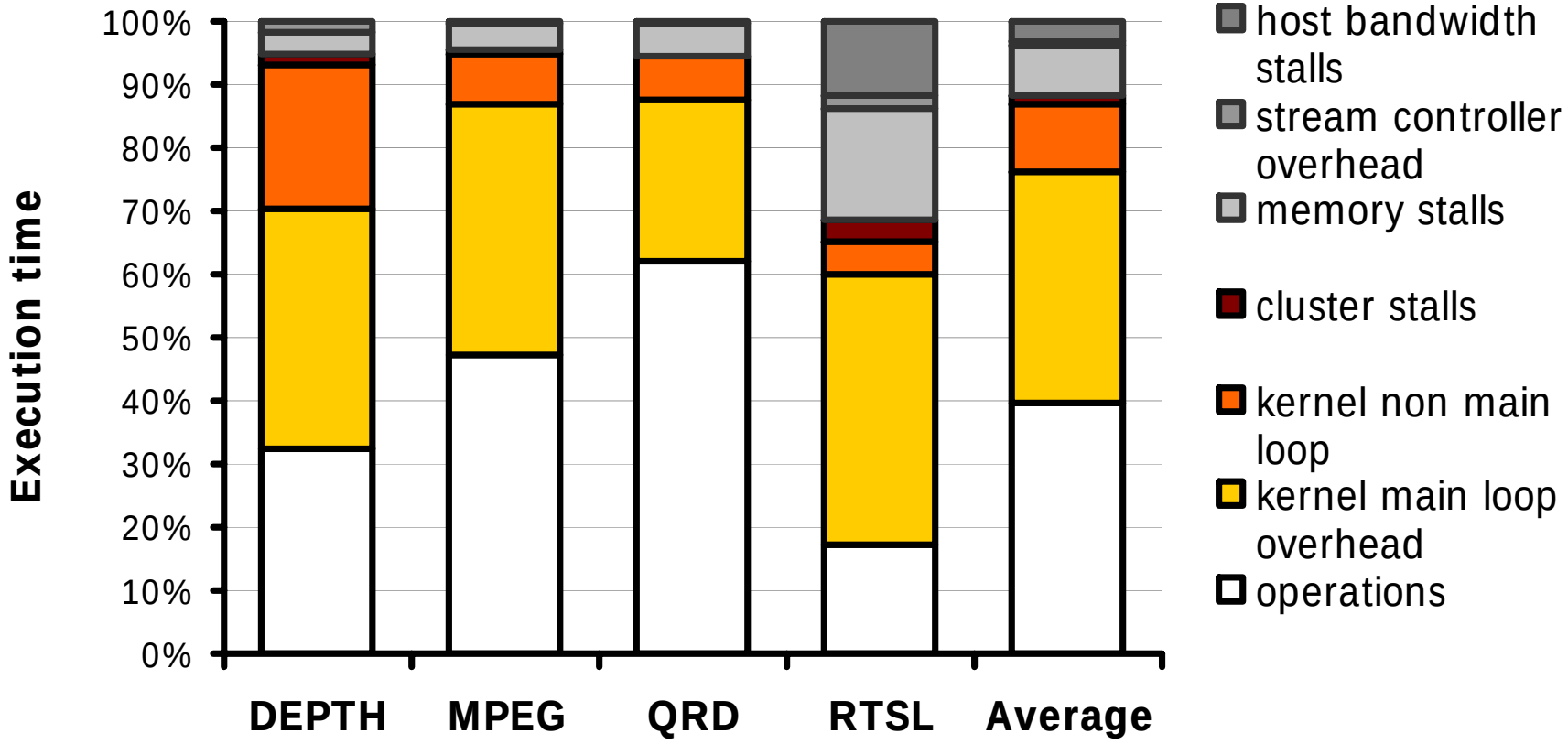




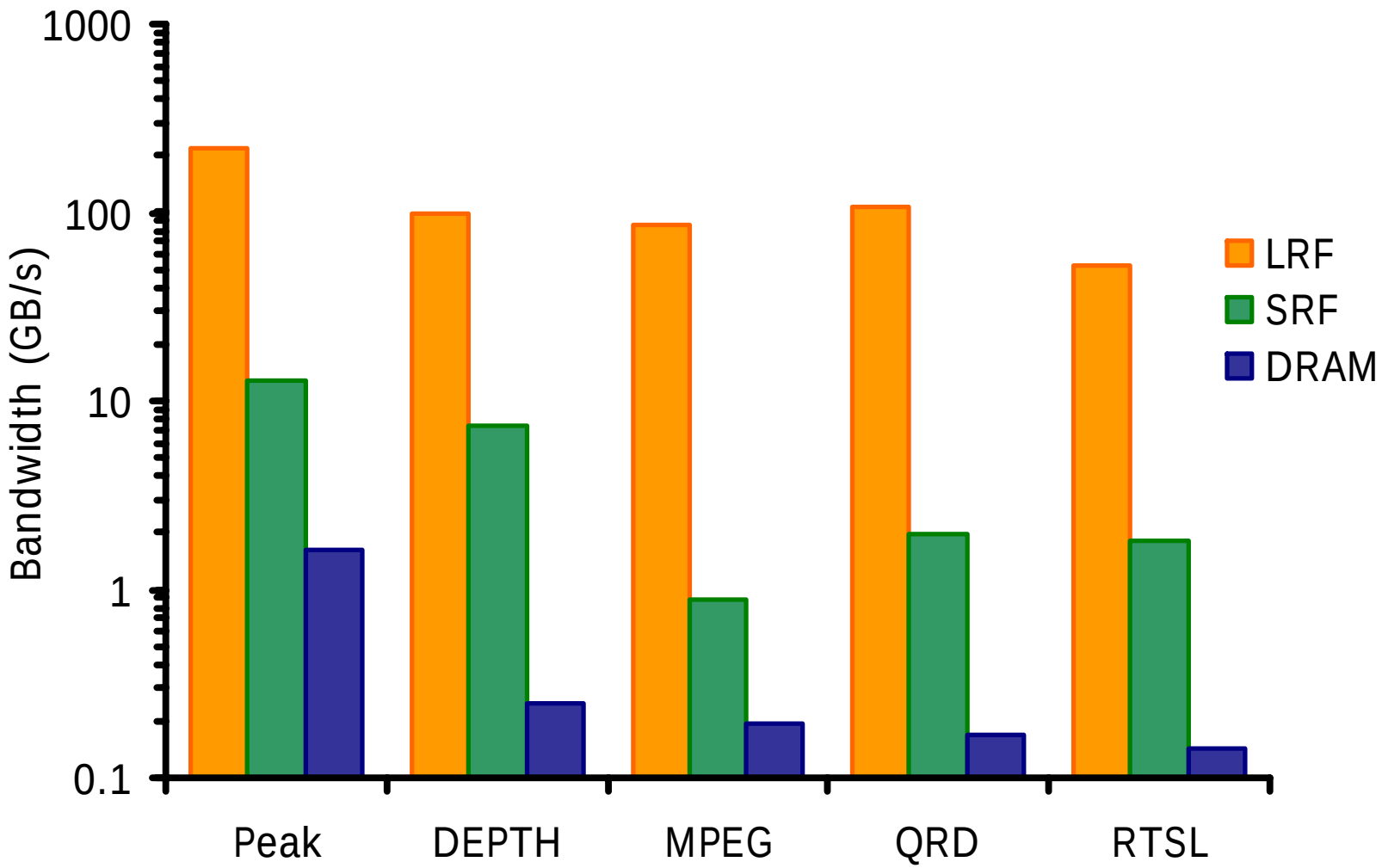
- Dual-Imagine test board



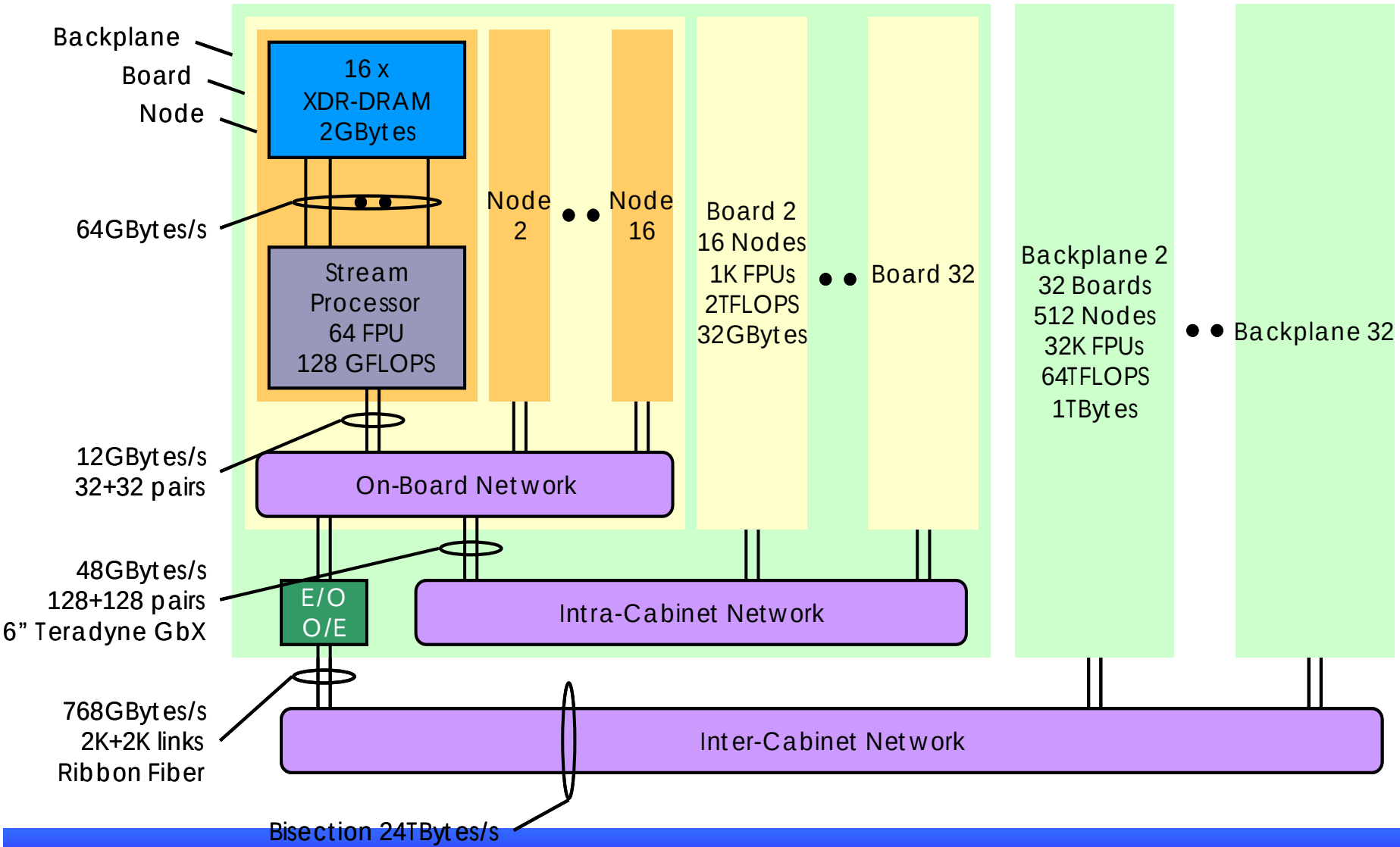
Application Performance (cont.)



Applications match the bandwidth hierarchy



Merrimac – Streaming Supercomputer



Scalable from 2-TFLOP workstation to 2-PFLOP supercomputer

Merrimac Application Results

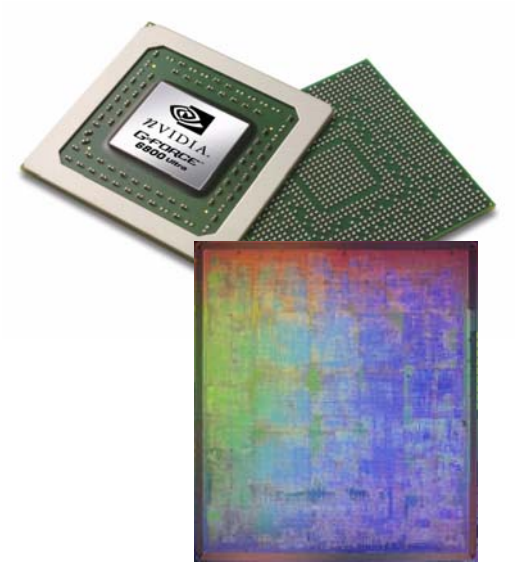
Application	Sustained GFLOPS	FP Ops / Mem Ref	LRF Refs	SRF Refs	Mem Refs
StreamFEM3D (Euler, quadratic)	31.6	17.1	153.0M (95.0%)	6.3M (3.9%)	1.8M (1.1%)
StreamFEM3D (MHD, constant)	39.2	13.8	186.5M (99.4%)	7.7M (0.4%)	2.8M (0.2%)
StreamMD (grid algorithm)	14.2*	12.1*	90.2M (97.5%)	1.6M (1.7%)	0.7M (0.8%)
GROMACS	38.8*	9.7*	108M (95.0%)	4.2M (2.9%)	1.5M (1.3%)
StreamFLO	12.9*	7.4*	234.3M (95.7%)	7.2M (2.9%)	3.4M (1.4%)

Simulated on a machine with 64GFLOPS peak performance and no fused MADD
* The low numbers are a result of many divide and square-root operations

Applications achieve high performance and make good use of the bandwidth hierarchy

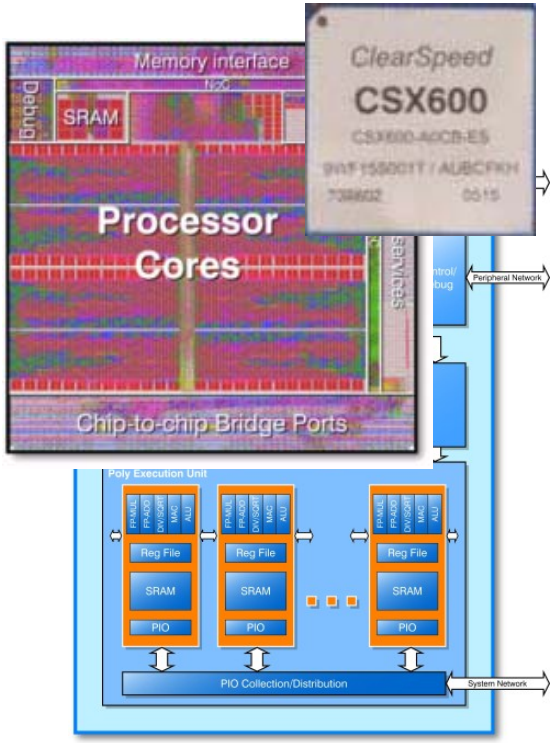
Other Stream Processors

Other Stream Processors



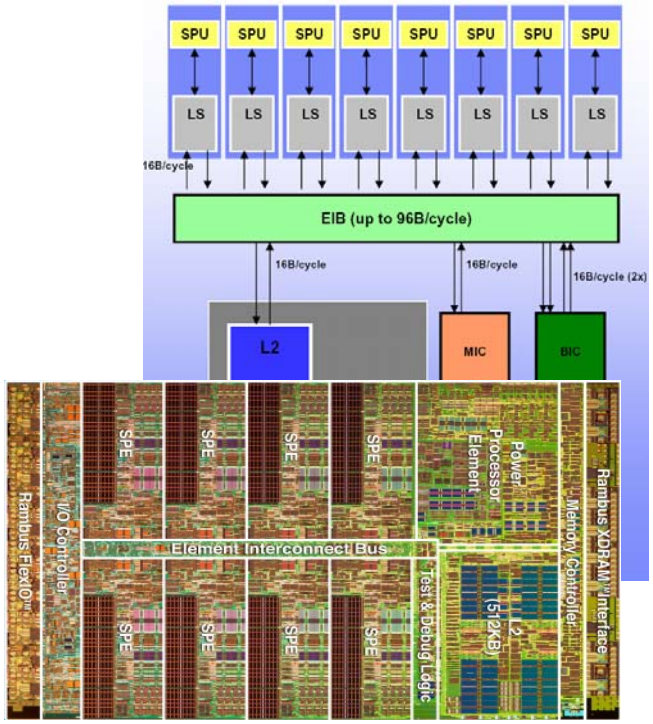
GPUs

50-100 GFLOP/s **10-30W**



ClearSpeed CSX600

96 GFLOP/s **10W**



STI Cell

~200 GFLOP/s **100W**

Other Stream Processors

- Technology pushing many to build stream processors
- GPUs (Nvidia, ATI), Game Processors (Cell), Physics Processors (Ageia), Accelerators (Clearspeed)
- Many (10s-100s) of FPUs
- Distributed local storage
- Latency hiding on access to external memory
  - Block access or deeply multithreaded

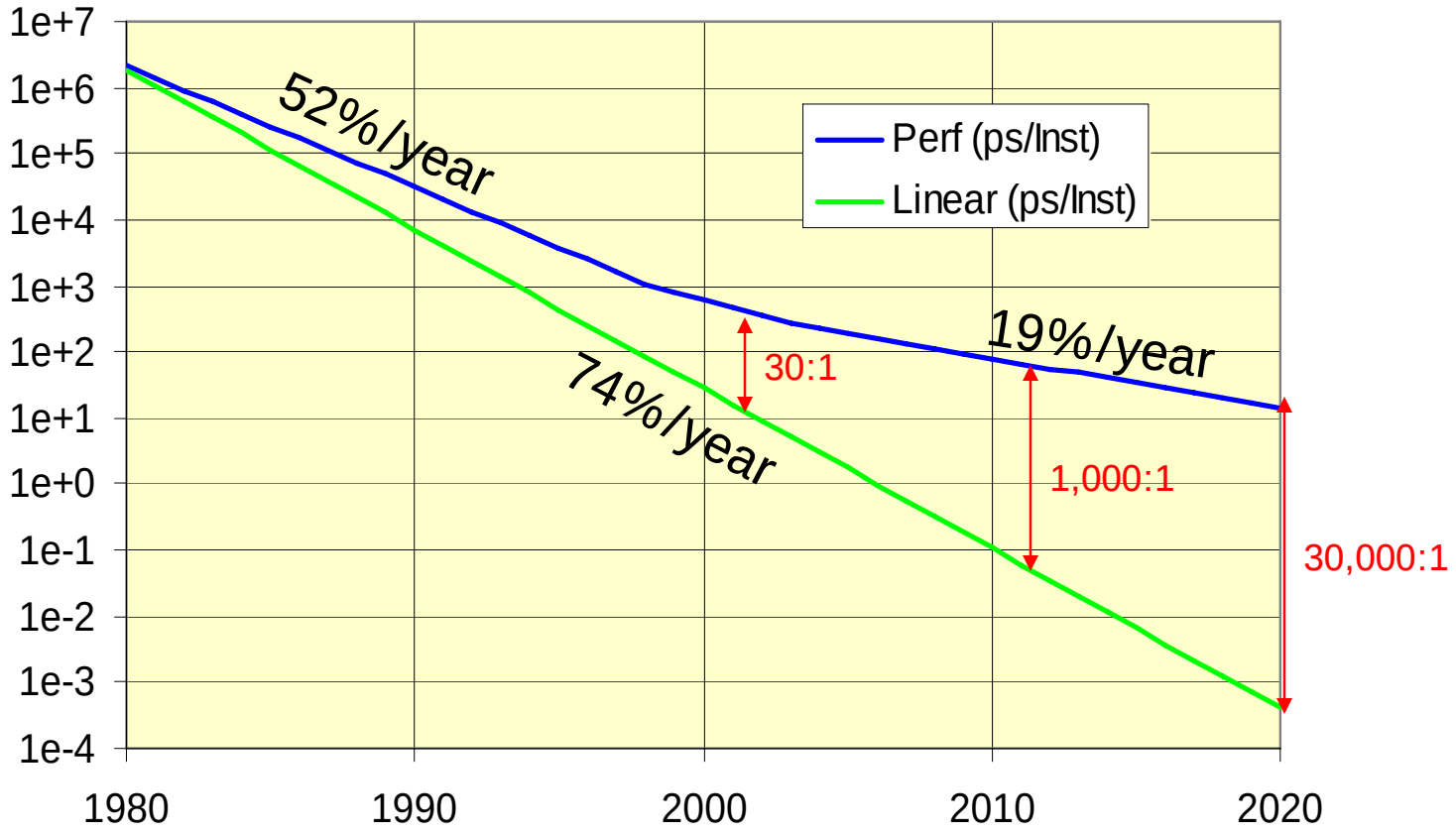
- All benefit from stream programming
  - But the right architecture makes it easier and more efficient

Architecture Issues

- On-chip memory
  - Read and ***write*** access to on-chip storage
    - Producer-consumer locality demands write access
  - Data movement between on-chip memories
    - Without going off chip
- Off-chip memory
  - No substitute for bandwidth
  - Efficient gather and scatter required

What’s Next?

ILP is mined out – end of superscalar processors Time for a new architecture



Dally et al. “The Last Classsical Computer”, ISAT Study, 2001

Computing landscape is changing

- Many function units
- Deep, distributed storage hierarchy
- Communication limited

- **Research is needed** to understand how to architect and program these processors

- Not an incremental fix:
  - Fundamental rethinking of basic architecture, programming model, and compilers is required

Software Topics

- Exposed Communication Programming Models
  - Abstract storage hierarchy and communication costs
  - Portable codes with predictable performance
- Compiling bulk operations
  - Strategic (vs tactical) program reorganization
  - Scheduling bulk data transfers
  - Size and shape of blocking
  - Irregular computations
    - Localization of shared neighbors
  - Variable size results
- Applications
  - Communication-efficient algorithms

Hardware Topics

- Close efficiency gap with hard-wired engines
  - Gap is 10-100x today (10 for stream processors)
  - Efficient data movement is first step
  - Other overheads remain to be removed
- Storage hierarchies that can be abstracted
- Balancing parallelism ILP x DLP x TLP
- On-chip networks
  - To connect within and between levels of the hierarchy
- Communication and synchronization mechanisms
  - Drives granularity – which in turn determines available parallelism
- Mechanisms for reuse of irregular data

Summary

- Communication is expensive, arithmetic is cheap
  - Parallelism to exploit arithmetic
  - Locality to conserve bandwidth
- Architectures evolving toward a deep, broad storage hierarchy
  - Storage to hide latency, cover bandwidth taper
  - Stream processors >10x efficiency of conventional CPUs
- Explicitly manage this hierarchy
  - Makes efficient use of scarce, expensive resources
  - Enables optimization
- Generalized Stream programming
  - Bulk operations: data movement and kernels
  - Parallelism, Locality, and Predictability