The Merrimac Project:
towards a Petaflop computer

Massimiliano Fatica
Stanford University

CASPUR
June 23 2003

Outline

- Motivation
- Overview of the Merrimac project
  - Hardware
  - Software
- Applications: StreamFLO
- Brooktran
- Conclusions

Motivation

From Teraflops to Petaflops

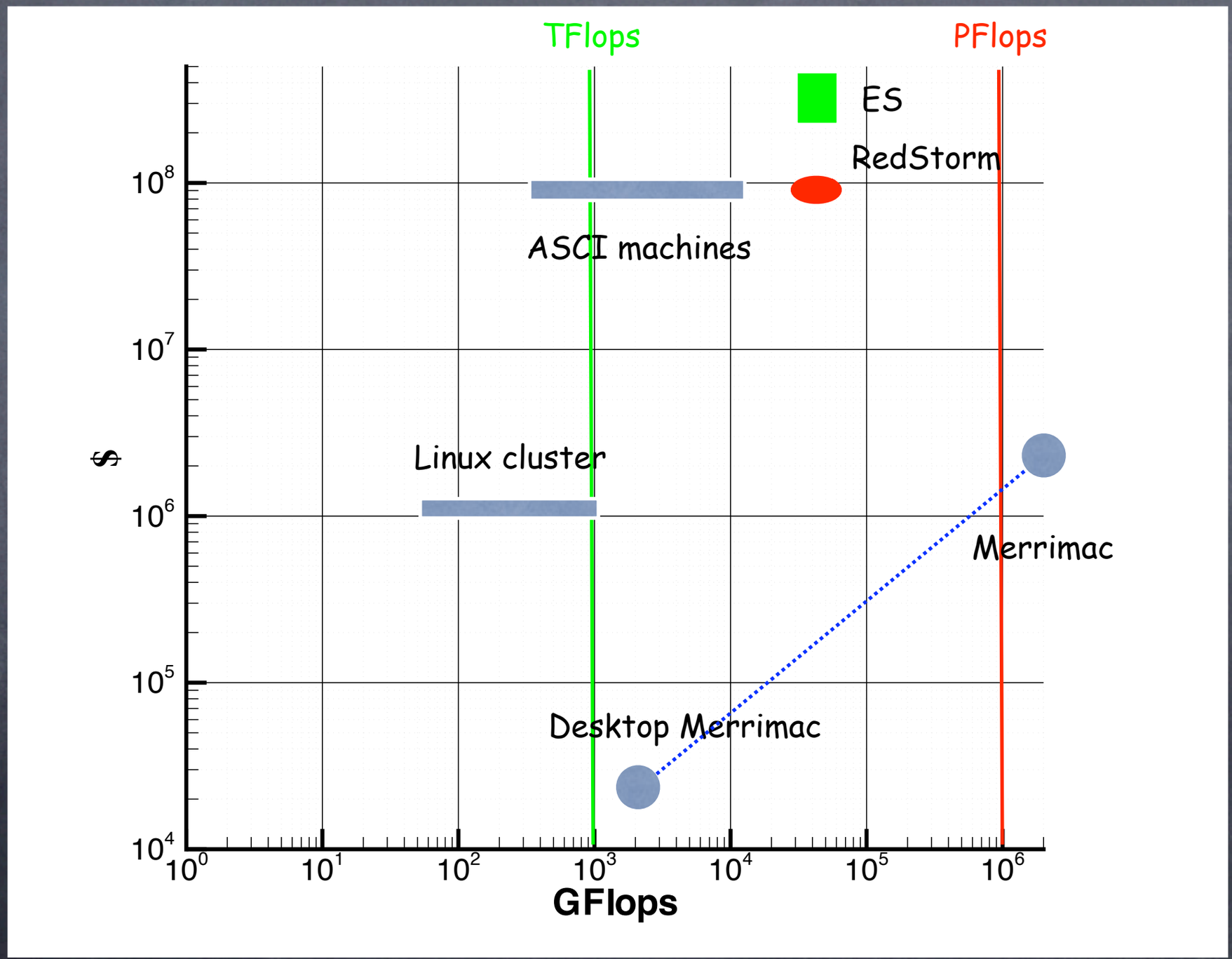
How to make the next step:

- Special purpose hardware:
  - Grape (gravitional N-body probelm)
  - MD-Grape (Molecular dynamics)
- General purpose hardware:
  - Keep building "monster" clusters
  - New architectures:
    - HTMT Petaflop project (lead by Sterling): use of esoteric technology, multithreaded architecture
    - BlueGene/L
    - Merrimac project: streaming supercomputer

Performance

- Theoretical peak performance of the ASCI machines are in the Teraflops range, but sustained performance with real applications is far from the peak
  - Salinas, one of the 2002 Gordon Bell Awards, was able to sustain 1.16 Tflops on ASCI White (less than 10% of peak)
- On the Earth Simulator, a custom engineered system with exceptional memory bandwidth, interconnect performance and vector processing capabilities
  - Global atmospheric simulation was able to achieve 65% of the 40 Tflops of peak performance

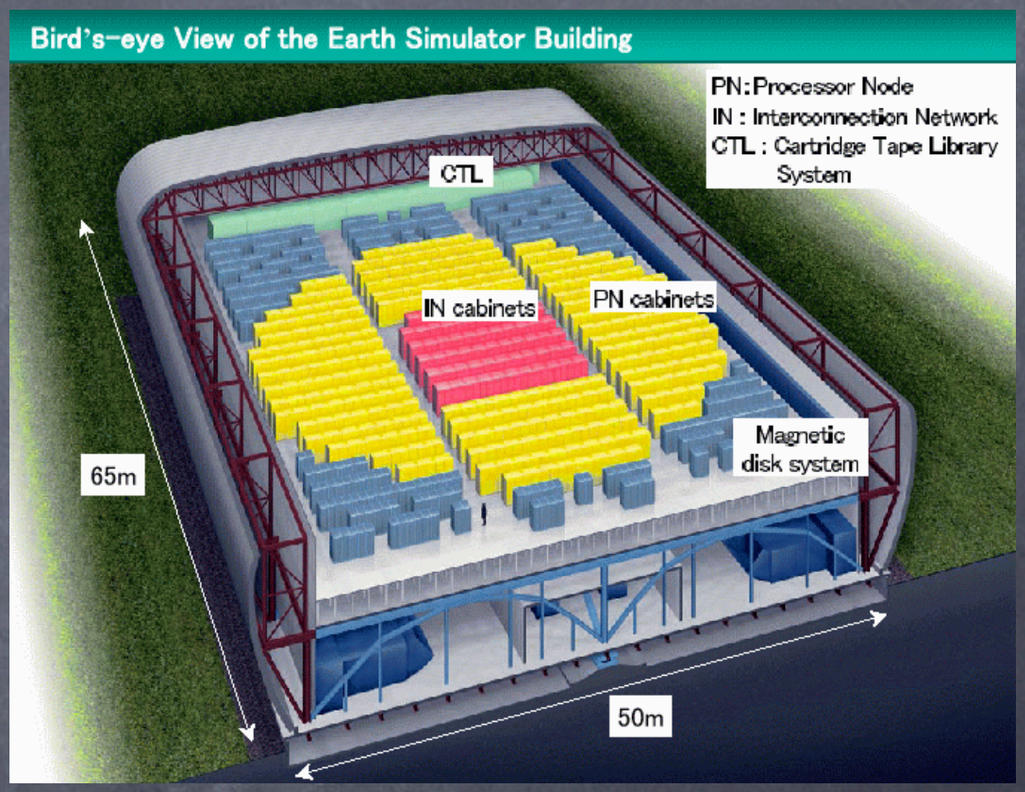
Price/Performance



Performance/Cost Comparisons

- Earth Simulator (today)
  - Peak 40TFLOPS, ~$450M
  - 0.09 MFLOPS/$
  - Sustained 0.04MFLOPS/$
- Red Storm (2004)
  - Peak 40TFLOPS, ~$90M
  - 0.44 MFLOPS/$
- Merrimac (proposed 2006)
  - Peak 2 PFLOPS: 40 TFLOPS, < $1M ($312K)
  - 128 MFLOPS/$
  - Sustained 64 MFLOPS/$ (single node)

Numbers are sketchy today, but even if we are off by 2x, improvement over status quo is large



Merrimac project

The Merrimac Team

- HARDWARE: B. Dally

  I. Ahn, A. Das, M. Horowitz, U. Kapasi, N. Jayasena

- SOFTWARE: P. Hanrahan

  I. Buck, M. Erez, J. Gummarju, T. Knight, C. Kozyrakis, F. Labonte, M. Rosemblum

- APPLICATIONS: J. Alonso

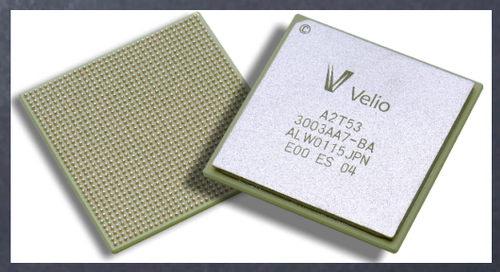
  T. Barth, E. Darve, M. Fatica, R. Fedkiw, F. Losasso, A. Wray

How did we achieve that?

VLSI Makes Computation Plentiful

- Abundant, inexpensive arithmetic
  - Can put 100s of 64-bit ALUs on a chip
  - 20pJ per FP operation
- (Relatively) high off-chip bandwidth
  - 1Tb/s demonstrated, 2nJ per word off chip
- Memory is inexpensive $100/Gbyte

nVidia GeForce4
~120 Gflops/sec
~1.2 Tops/sec

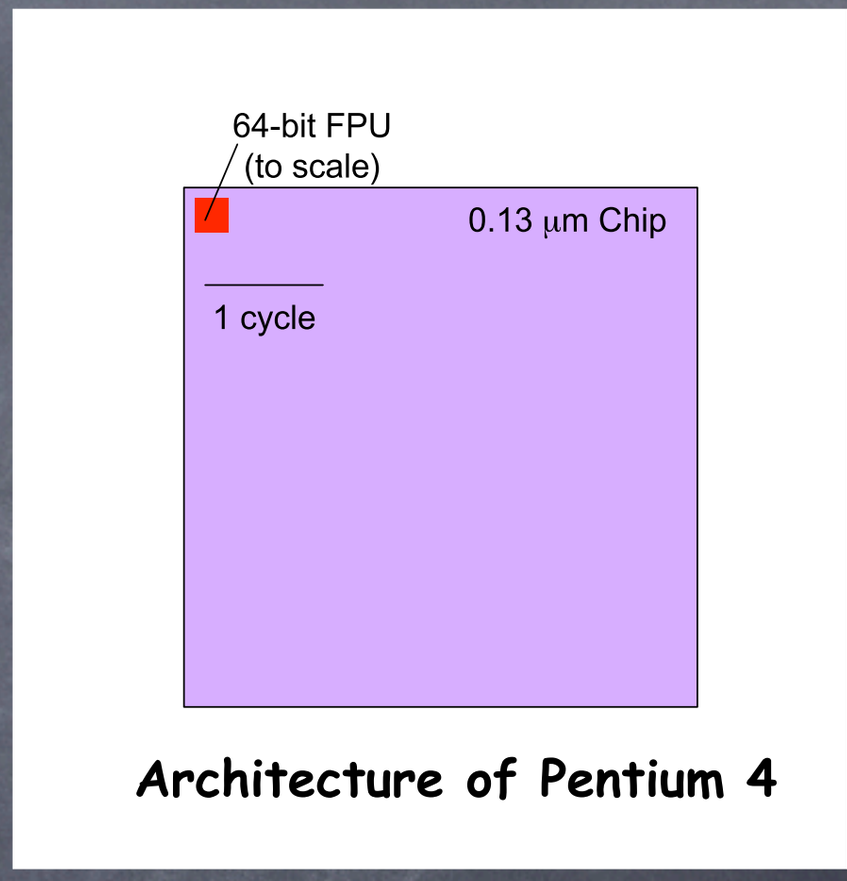


Velio VC3003
1Tb/s I/O BW

Exploit VLSI technology

Objectives for the Streaming architecture:

- Parallelism:
  - To keep 100s of ALUs per chip (thousands/ board millions/system) busy
- Locality of data:
  - To match 20Tb/s ALU bandwidth to ~100Gb/s chip bandwidth.
- Latency tolerance:
  - To cover 500 cycle remote memory access time



Architecture of Pentium 4

Current Architecture: few ALUs / chip = expensive and limited performance.

Arithmetic is cheap, global bandwidth is expensive
Local << global on-chip << off-chip << global system

Benefits of the streaming architecture

Streaming scientific computations exploits the capabilities of VLSI

- Modern VLSI technology makes arithmetic cheap
  - 100s of GFLOPS/chip today, TFLOPS in 2010
- But bandwidth is expensive
- Streams change the ratio of arithmetic to bandwidth
  - By exposing producer-consumer locality
    - Cannot be exploited by caches – no reuse, no spatial locality
- Streams also expose parallelism
  - To keep 100s of FPUs per processor busy
- High-radix networks reduce cost of bandwidth when its needed
  - Simplifies programming

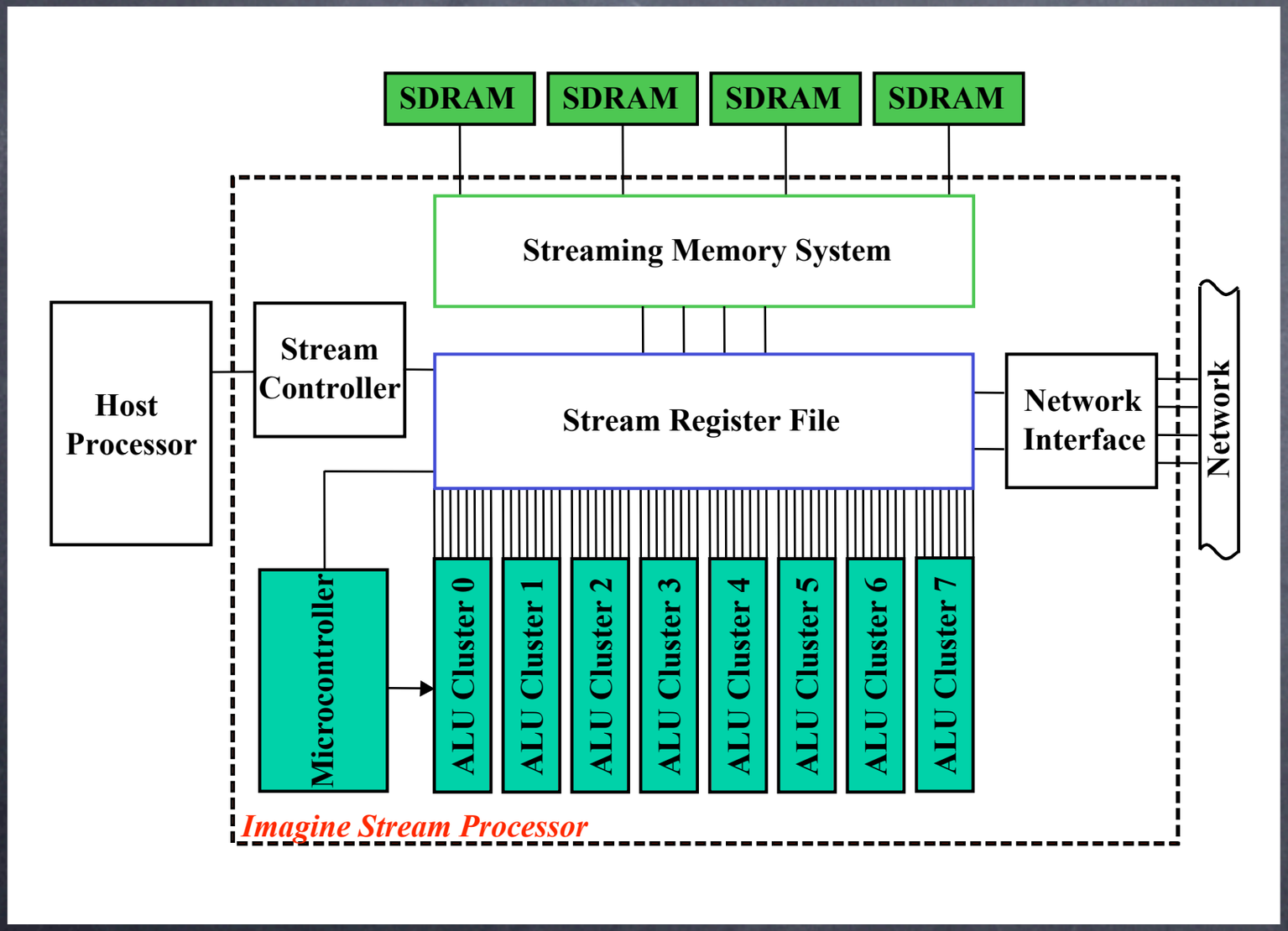
Merrimac project

- Stream processors as a building block
- High-performance interconnection to provides good global bandwidth
- New programming paradigm to exploit this new architecture:
  - Strong interaction between application developers and language development group
- It will scale from a 2 Tflops workstation to a 2 Pflops machine with 16K processors

Merrimac project

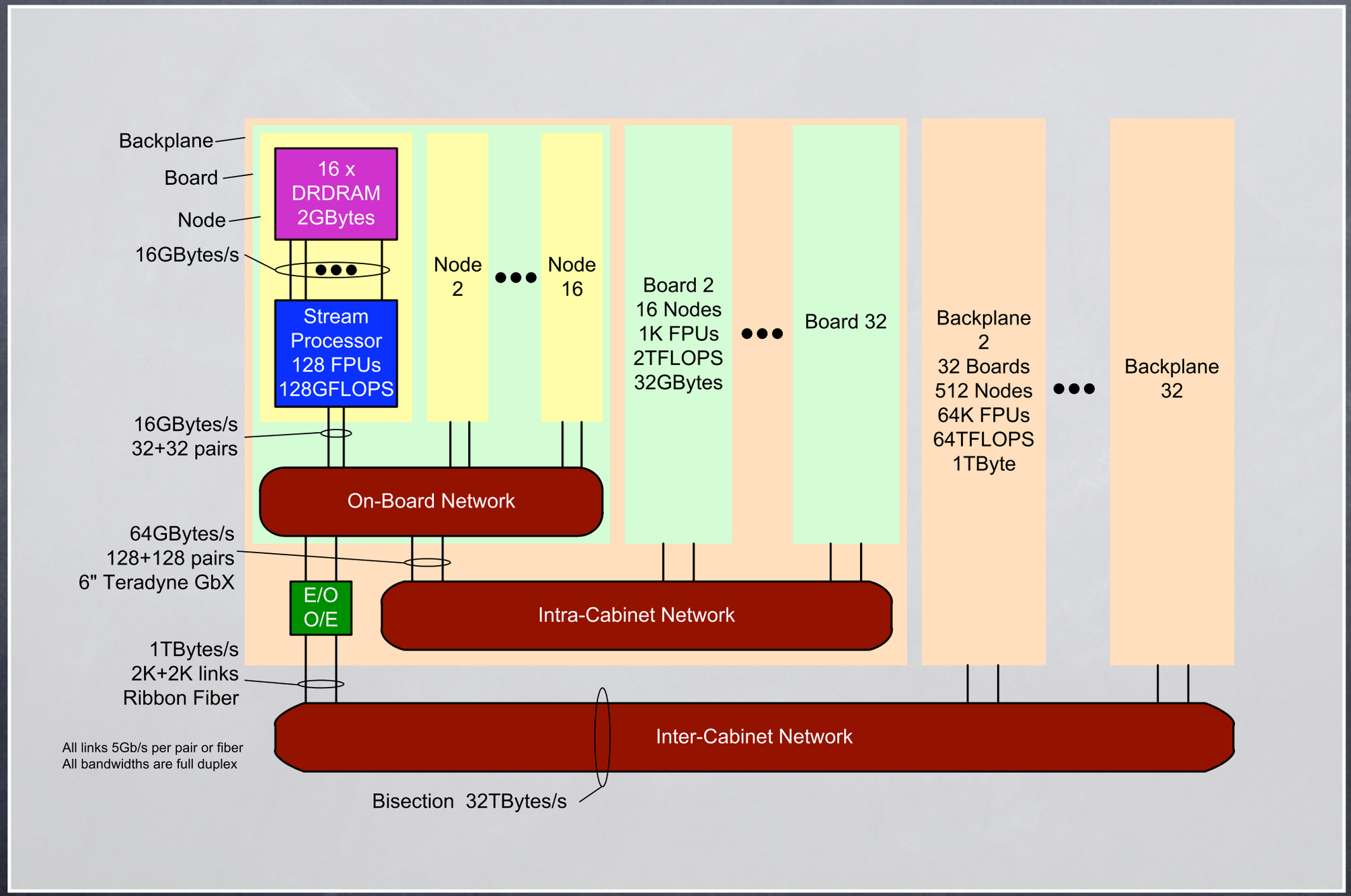
Hardware

The Imagine Stream Processor

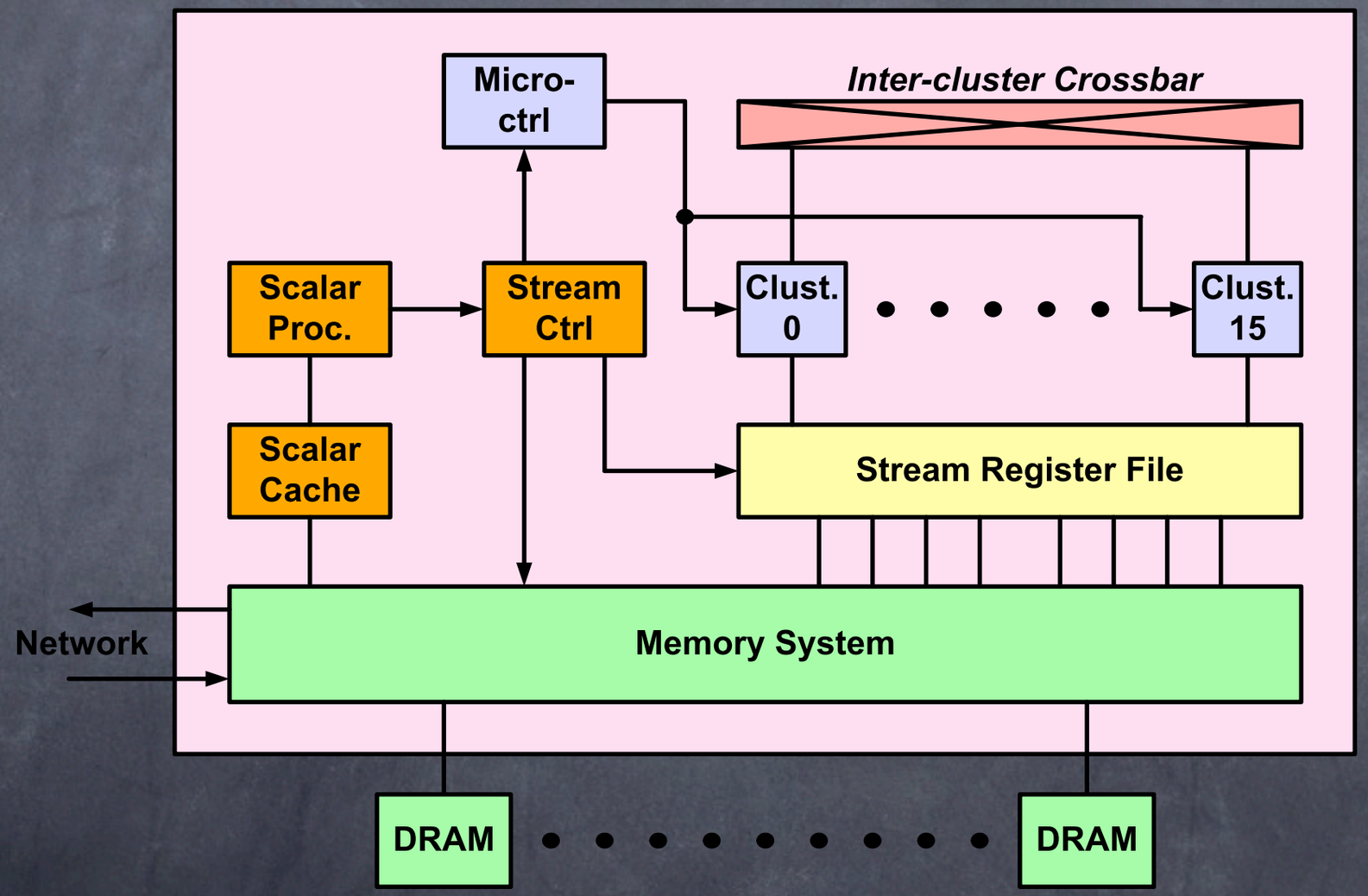


- Programmable signal and image processor
- Peak performance of 20 GFlops (32 bit), sustains over 12 GFlops on key signal processing benchmarks
- Has shown the potentiality of the streaming architecture
- It is not easy to program (Kernel-C, Stream-C)

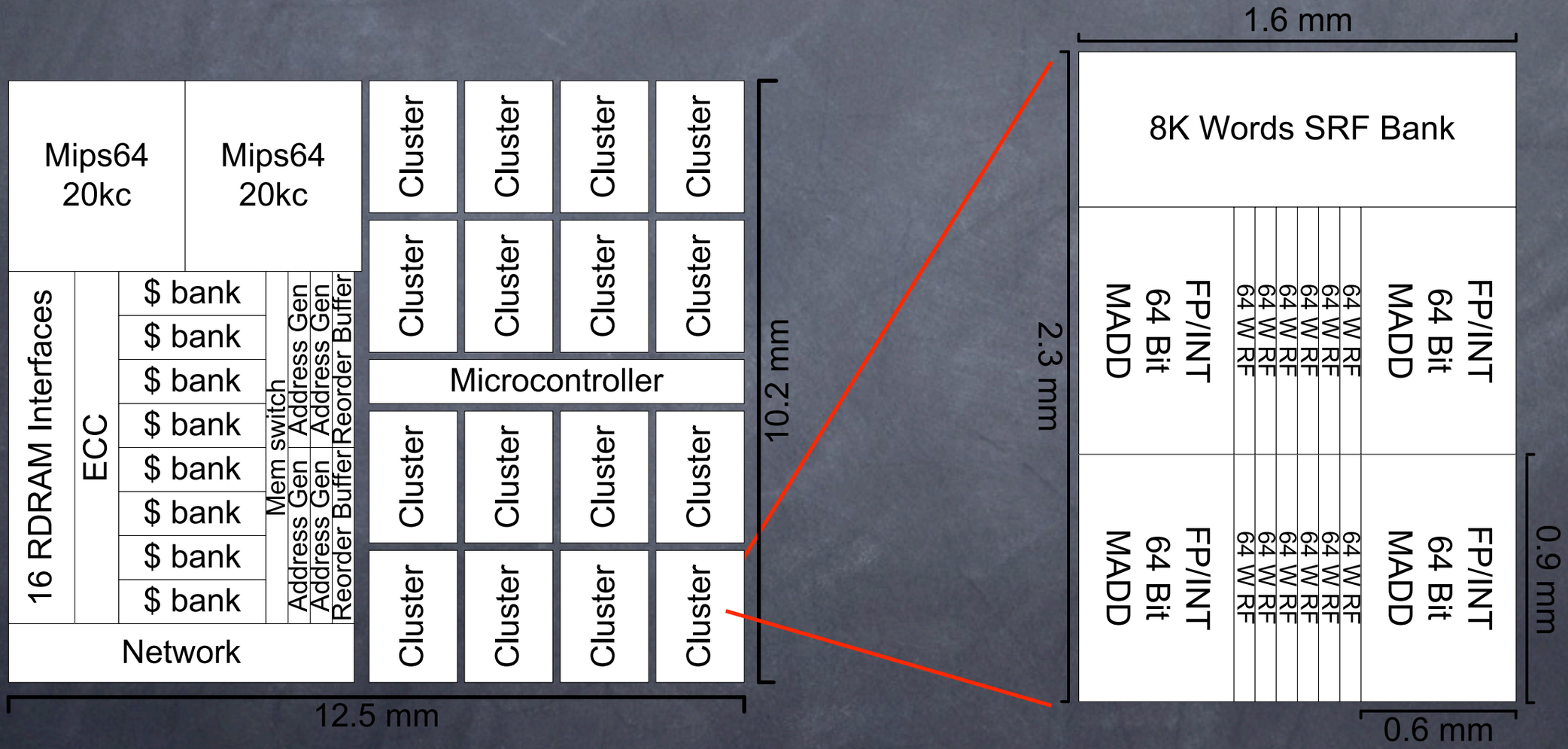
Merrimac Architecture



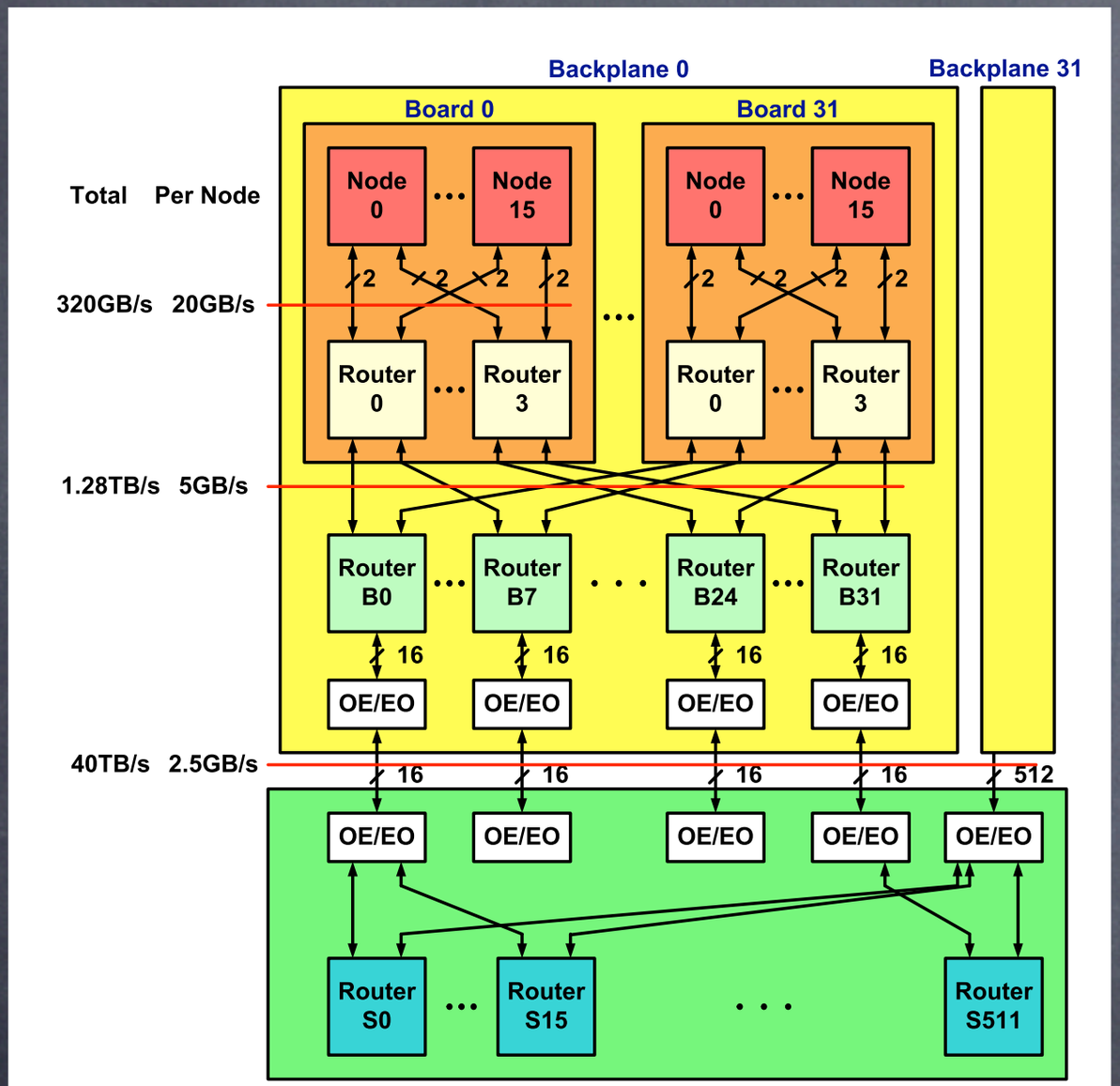
Merrimac node



Merrimac stream processor chip



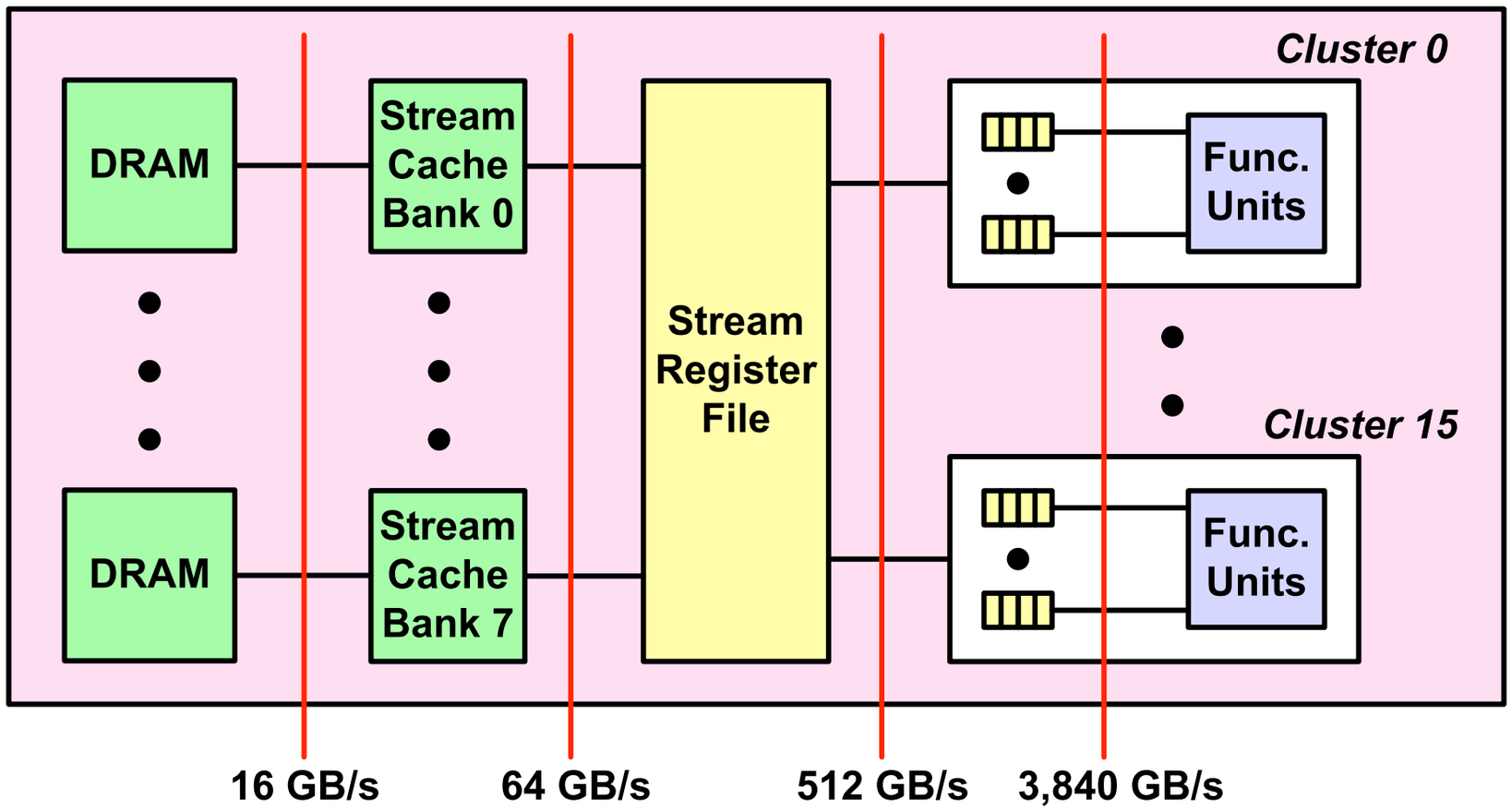
Merrimac network



- Flat memory bandwidth within a 16-node board
- 4:1 Concentration within a 32-node backplane, 8:1 across a 32 backplane system
- Routers with bandwidth B=640Gb/s route messages with length L=128b
  - Requires high radix to exploit

High radix routers enable economical global memory

Merrimac Bandwidth



Bandwidth taper matches capabilities of VLSI to demands of scientific applications

Bandwidth hierarchy enabled by streams

Level	Bandwidth per Node (GB/s)
Cluster registers	3,840
Stream register file	512
Stream cache	64
Local Memory	16
Board Memory (16 Nodes)	16
Cabinet Memory (1K Nodes)	4
Global Memory (16K Nodes)	2

Streaming

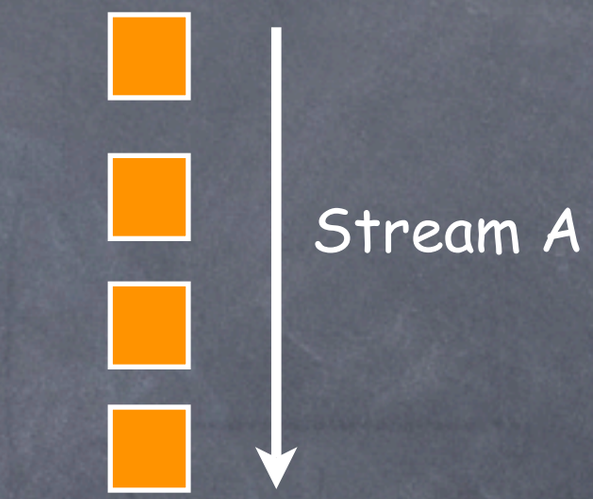
Network

Merrimac project

Software

Brook: streaming language

- C with streaming construct:
  - Make data parallelism explicit
    - Declare communication pattern
- Streams:
  - Streams are views of memory
  - Records operated on in parallel



Brook kernels

Kernels are functions which operate only on streams

- Streams arguments are read-only or write-only
- Special reduction variables
- No "state" or static variables allowed
- No global memory access

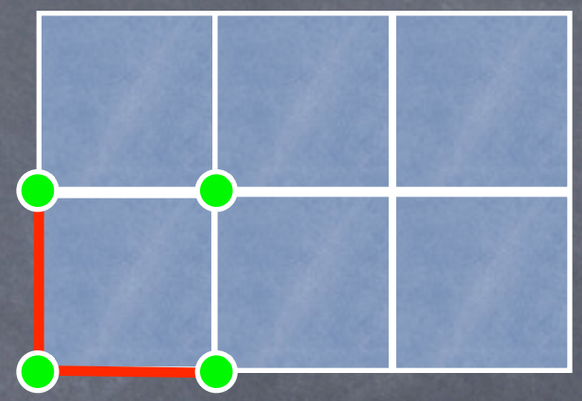
How to write a stream application?

- To port a Fortran/C code to Brook:
  - define the data layout and access patterns
  - Define the computation on data
- Decouple the data access pattern from the computation
- Code is more clean and easy to understand

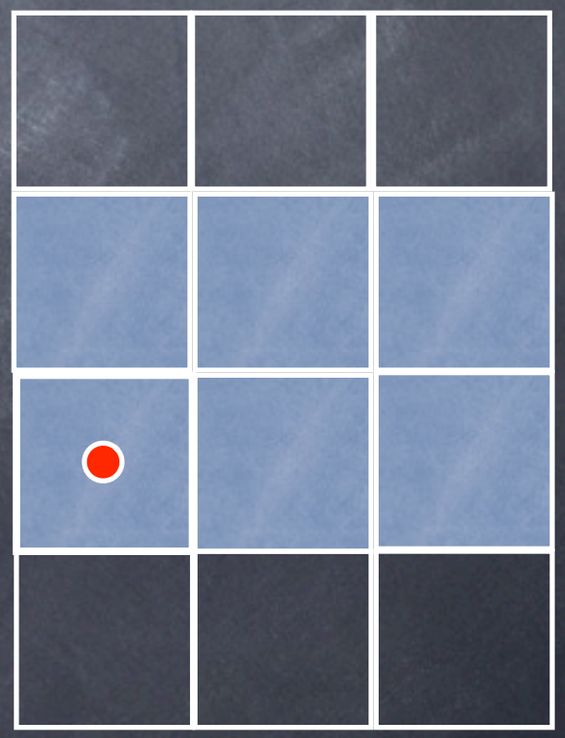
Brook Example

```
struct Grid_struct{ float x; float y;};
typedef stream struct Grid_struct Gridstream;
typedef struct Grid_struct Gridarray;
typedef stream float Floats;
typedef stream Gridstream **grid2d_s;

int main(int argc, char** argv) {
Gridarray*  mesh; float* vol;
Gridstream meshstream; Floats volstream;
gird2d_s grid2d “2,2”;
int nx=3; ny=2
mesh=malloc(sizeof(Gridarray)*(nx+1)*(ny+1));
vol=malloc(sizeof(float)*nx*(ny+2));
.....
call streamLoad(meshstream,mesh,(nx+1)*(ny+1));
call streamShape(meshstream,meshstream,2,nx+1,ny+1);
call streamStencil(grid2d,meshstream,STREAM_STENCIL_HALO,2,0,1,0,1);
call ComputeMetric(grid2d,volstream,&volmin,&volmax);
call streamStore(volstream,vol+nx,nx,ny);
.....}

kernel void ComputeMetric(grid2d_s grid,out floats volume,
              reduce float *volmin, reduce float *volmax) {
 volume=.5*((grid[1][1].x-grid[0][0].x)*(grid[0][1].y-grid[1][0].y)
               -(grid[1][1].y-grid[0][0].y)*(grid[0][1].x-grid[1][0].x));
*volmin  = *volmin < volume ? *volmin : volume;
*volmax = *volmax > volume? *volmax : volume; }
```

mesh



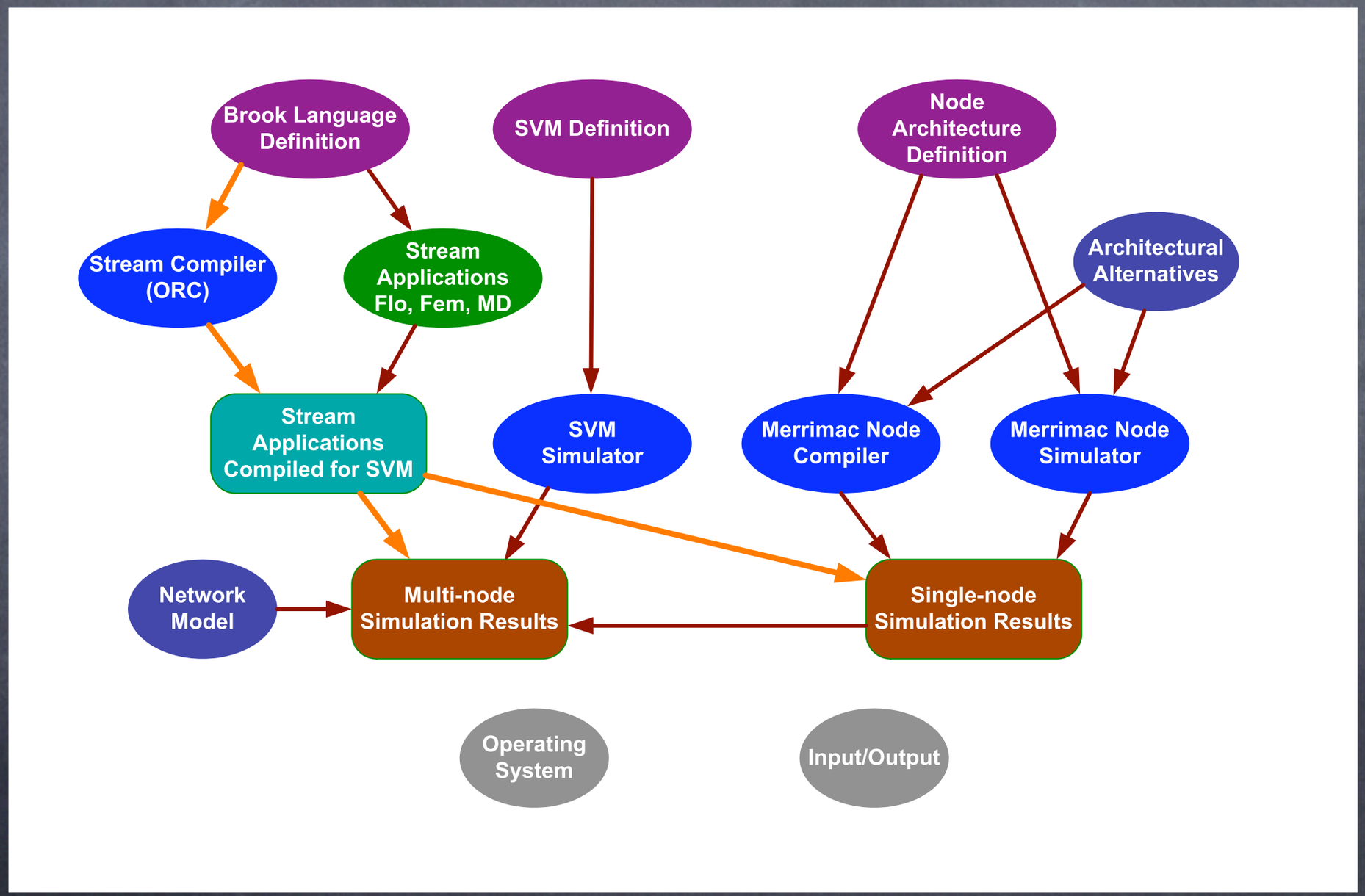
vol



Brook compilers

- There are two Brook implementations:
  1. compiler based on the Metacompiler, that translates Brook to C and uses run-time library: it is used to develop and debug codes.
  2. compiler based on the Imagine tools: it used to assess performance and to study different hardware configurations
- A new compiler infrastructure is being developed using the Open64 compiler

Major tasks


Brook Language Definition
SVM Definition
Node Architecture Definition
Stream Compiler (ORC)
Stream Applications Flo, Fem, MD
Architectural Alternatives
Stream Applications Compiled for SVM
SVM Simulator
Merrimac Node Compiler
Merrimac Node Simulator
Network Model
Multi-node Simulation Results
Single-node Simulation Results
Operating System
Input/Output

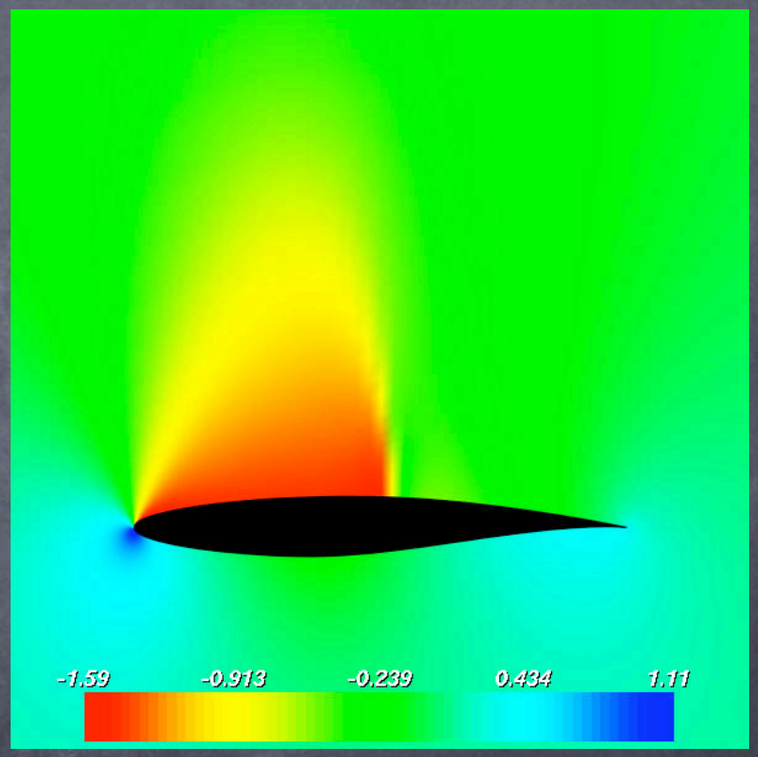
Applications

- 3 major applications:
  - StreamFEM
  - SteamFLO
  - StreamMD

StreamFLO

StreamFLO

- StreamFLo is a streaming version of FLO82 [Jameson] for the solution of the inviscid flow around an airfoil



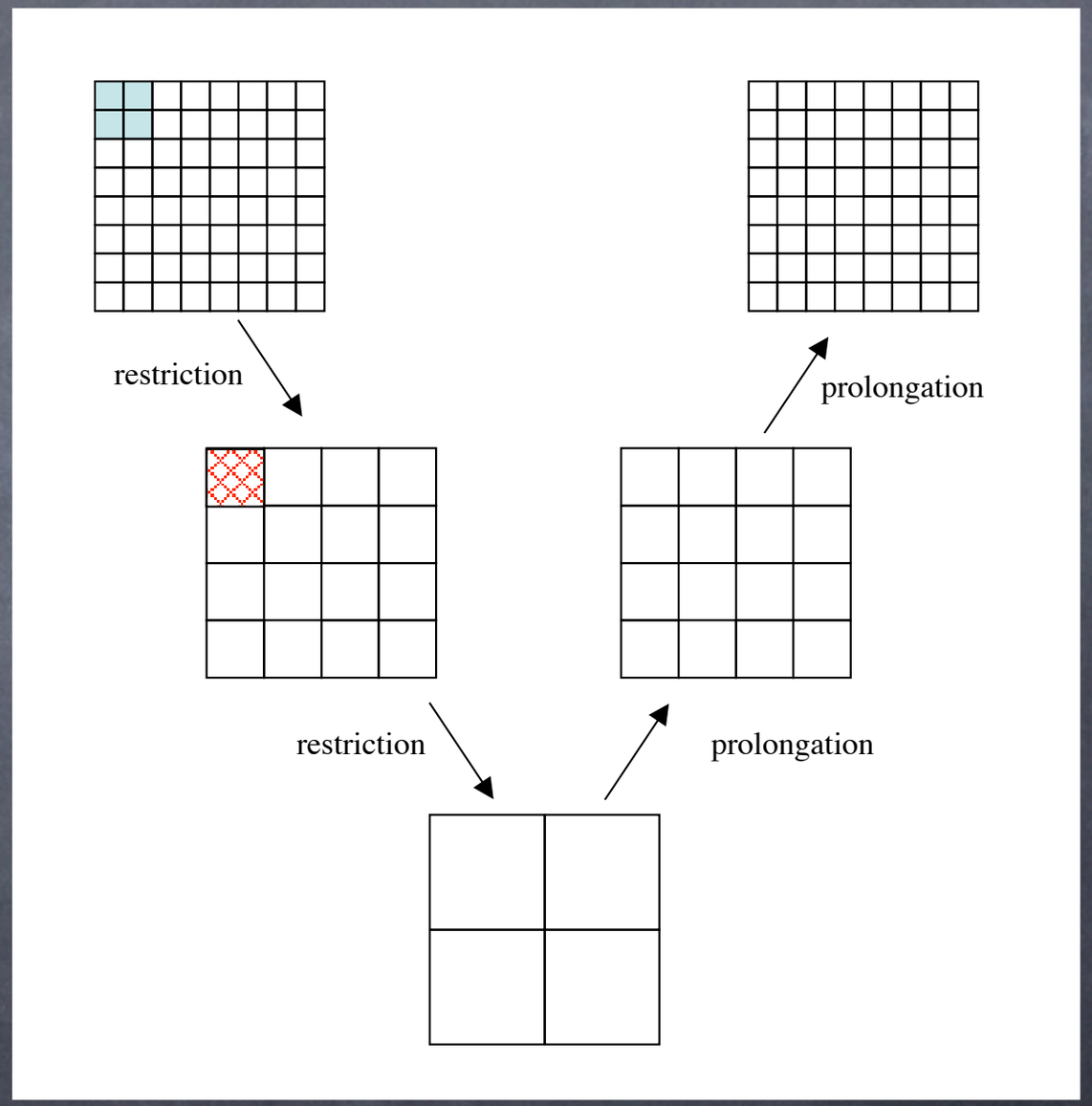
- The code uses a cell centered finite volume formulation with a multigrid acceleration to solve the 2D Euler equations

- The structure of the code is similar to TFLO and the algorithm is found in many compressible solvers

Code organization

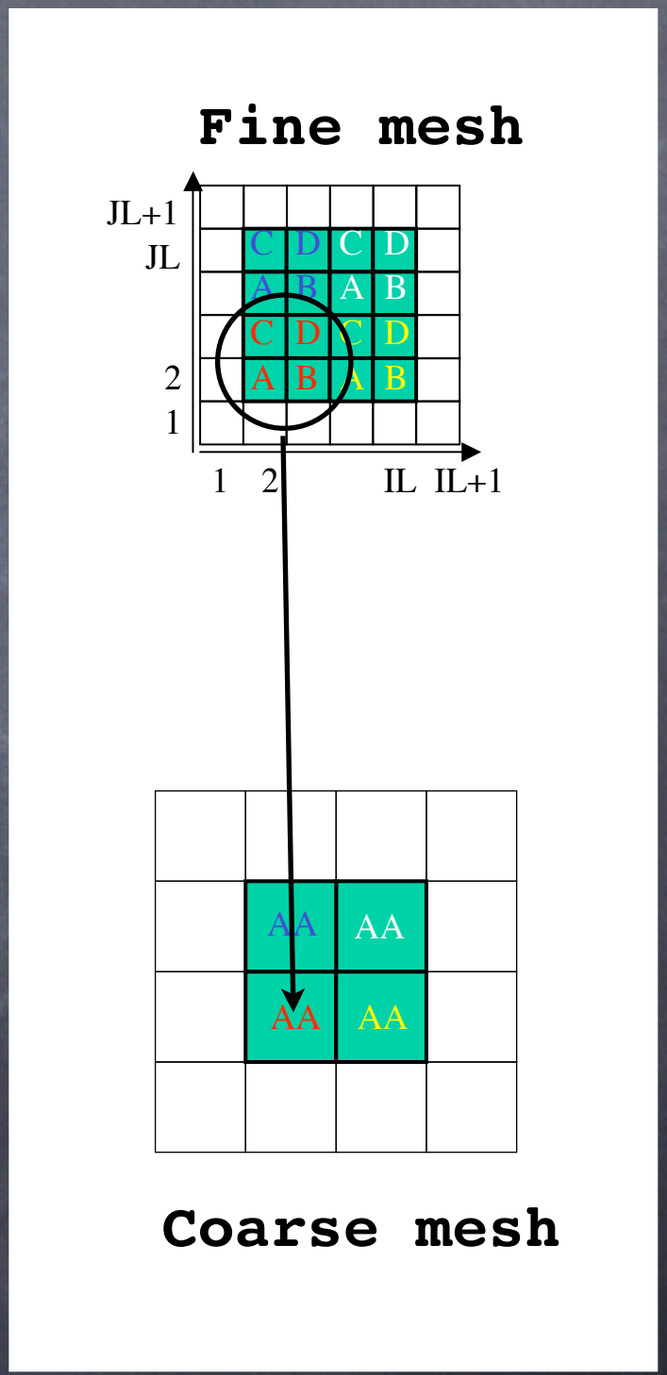
- There is an external driver that controls the multigrid strategy and the multigrid data movement (restriction and prolongation). All the multigrid levels are stored in a 1D array.

- On each multigrid level, the code operates on 2D grids where it needs to compute the time step. Most of the operations are performed on stencil (3x3 and 5x5)



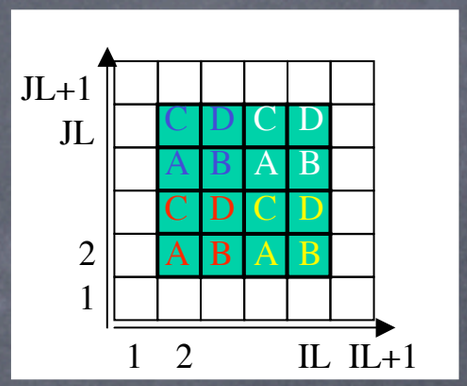
Original FORTRAN subroutine

```
C   ***************************************************************
C   *                                                             *
C   *   TRANSFERS THE SOLUTION  TO A COARSER MESH                 *
C   *                                                             *
C   ***************************************************************
      ..........
      DO  N=1,4
        JJ      = 1
        DO  J=2,JL,2
          JJ      = JJ  +1
          II      = 1
          DO  I=2,IL,2
          II      = II  +1
         WWR(II,JJ,N)  =
                   (DW(I,J,N)*VOL(I,J) +DW(I+1,J,N)*VOL(I+1,J)
            .      +DW(I,J+1,N)*VOL(I,J+1) +DW(I+1,J+1,N) *VOL(I+1,J+1))/
            .      (VOL(I,J)+VOL(I+1,J)+VOL(I,J+1)+ VOL(I+1,J+1))
           END DO
         END DO
      END DO
      ..........
```



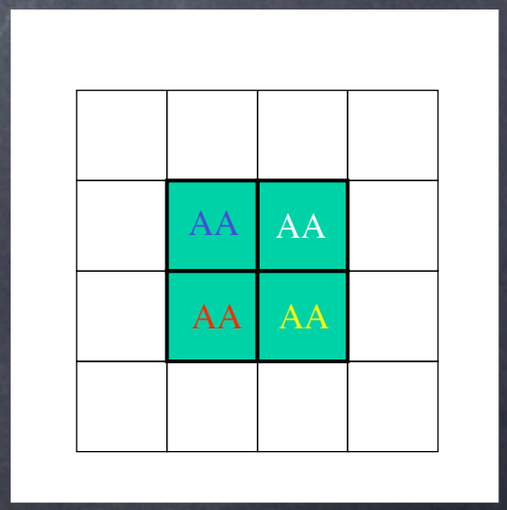
Equivalent Brook Code

Define access pattern:




```
// Fine mesh: flow  is a 2D stream of shape (nx+2,ny+2)
// Coarse mesh: coarse_flow is a 2D stream of shape (nx/2,ny/2)
// Select interior points on the fine mesh
 streamDomain(flow,flow,2,2,nx+1,2,ny+1)
 streamDomain(vol,vol,2,2,nx+1,2,ny+1)
// Generate a stream with the groups {(A,B,C,D), (A,B,C,D), (A,B,C,D), (A,B,C,D)}
  streamGroup(local_flow2d,flow,STREAM_GROUP_HALO,2,2,2);
  streamGroup(local_vol2d,vol,STREAM_GROUP_HALO,2,2,2);
//Apply restriction operator to generate stream with (AA,AA,AA,AA)
  TransferFieldFineCoarse(local_flow2d, local_vol2d, coarse_flow)
```

Define computational kernel:




```
kernel void TransferFieldFineCoarse(flow2d_s fine_flow,float2d_s vol,
                                    out Flow coarse_flow)
{
   coarse_flow.rho=(vol[0][0]* fine_flow[0][0].rho + vol[0][1]* fine_flow[0][1].rho
                  + vol[1][0]* fine_flow[1][0].rho + vol[1][1]* fine_flow[1][1].rho) /
                    (vol[0][0]+ vol[0][1]+ vol[1][0]+ vol[1][1]);
.....}
```

Equivalent Brook code

Streams definitions:

```
struct  Flow_struct {
float rho;  /* density */
float u;    /* momentum in x direction=density*velocity_x */
float v;    /* momentum in y direction=density*velocity_y */
float e;    /* Total energy= density*entalpy-pressure */
float p;    /* pressure */
};

typedef stream struct Flow_struct Flow;
typedef stream float floats;
typedef stream Flow **flow2d_s;
typedef stream floats **float2d_s;

main(int argc, char** argv)
{
  Flow flow,local_flow,interior_flow,coarse_flow;
  flow2d_s local_flow2d "2,2";
 ……..
}
```

Preliminary performance:

kernel schedule, H-CUSP dissipative flux

This kernel currently runs at 37 GFlops on the Merrimac simulator[1]: 58% of the peak performances

Improving the performance of the stencil setup will bring the performance up to 50 GFlops

[1]The simulator does not yet support MADD instructions: the peak performance is 64 GFlops

Ongoing work

- Working on 3D version of StreamFLO
- Moving from Euler to Navier-Stokes
- Evaluate different strategies for flux computations

Brooktran

Fortran

- Why do we need Fortran support?
  - Most scientific and high performance codes are in Fortran
  - National labs, NASA, aerospace companies have a huge investment in Fortran codes
  - The codes have been thoroughly tested and validated
  - They can be HUGE
  - Even if rewriting a code in a different language is not be a big deal, the validation process is
  - A code not fully validated can be acceptable in academia but not for real missions.

Porting codes to Merrimac

The path from C to Brook is much easier than the one from Fortran to Brook:

- C to Brook: similar to OpenMP parallelization in extent of changes
  - Start from original code and, one by one, "streamify" functions.
  - You can start working on the time consuming part of the code
  - Very easy to check the results since all the I/O & utility functions are working
- Fortran to Brook: more extensive changes than even MPI parallelization
  - The code has to be rewritten from scratch
  - A lot of time must be spent rewriting I/O and utility functions
  - Checking the results and debugging is very time consuming

Possible paths from Fortran to Brook (1)

- Mixed language programming: Fortran+Brook
  - Use the original structure of the Fortran code.
  - The subroutines that are floating-point intensive are replaced by Brook kernels
  - Streams are a view of memory, so we just need to pass the proper memory information to Brook
  - It requires some "glue" code and a standard Fortran compiler

Example of mixed language programming

```
// FORTRAN main
program sample
real, allocatable, dimension(:):: a,b,c
integer:: n
n=1000
allocate(a(n),b(n),c(n))
call brook_sum(a,b,c,n)
end program sample

// Brook function
void brook_sum(a,b,c,n)
float a[],b[],c[];
int *n;
{
 floats stream_a, stream_b, stream_c;
 streamLoad(stream_a,a,n);
 streamLoad(stream_b,b,n);
 add_array(stream_a,stream_b, stream_c);
streamStore(stream_c,c,n);
}

// Brook kernel
kernel add_array( floats a, floats b, out floats c)
{ c=a+b; }
```

Possible paths from Fortran to Brook (2)

- Brooktran: streaming language that uses Fortran syntax
  - The setup of streams is done through library calls
  - The kernels are written using a Fortran syntax and have the same constraints as Brook kernels

Example of Brooktran

```
// FORTRAN main
program sample
real, allocatable, dimension(:):: a,b,c
stream, real, dimension(:) :: stream_a, stream_b, stream_c
integer:: n
n=1000
allocate(a(n),b(n),c(n))
call streamLoad(stream_a, a, n)
call streamLoad(stream_b, b, n)
call add_array(stream_a, stream_b, stream_c)
call StreamStore(stream_c,c,n)
end program sample

// Brooktran kernel
kernel subroutine add_array(a, b, c)
stream, real, intent(in):: a, b
stream, real, intent(out):: c
c=a+b
end subroutine add_array
```

Brooktran

- A Fortran syntax of Brook will help porting legacy codes to Merrimac
- Open64 already has a Fortran95 front-end
- Fortran 9x array syntax makes stream code very compact

Brooktran syntax

New keywords

- We need to add the following keywords to Fortran:
- stream: used to define a stream; it is a native compound object much like an array
- kernel: used to specify a function or subroutine that can be executed by the streaming processor unit
- reduce: used for reduction arguments in kernels

Kernel

- Kernel functions or subroutines are declared by placing the "kernel" keyword before the function or subroutine name
- Arguments of the call have the same restrictions as in Brook.
- All arguments need to have explicit "intents"

```
kernel subroutine streamsum( a ,b, c, sum)
stream, real, intent (in):: a,b
stream, real, intent (out):: c
real, intent(reduce):: sum
    c=a+b
    sum=sum+c
end  subroutine streamsum
```

Stream

- Streams are a native compound object like arrays
- The shape is defined by the dimension given in the declaration

```
stream, type(real), dimension(:,:):: a
```

For streams that are generated from stencil of group operators, we can specify the "shape" of each element

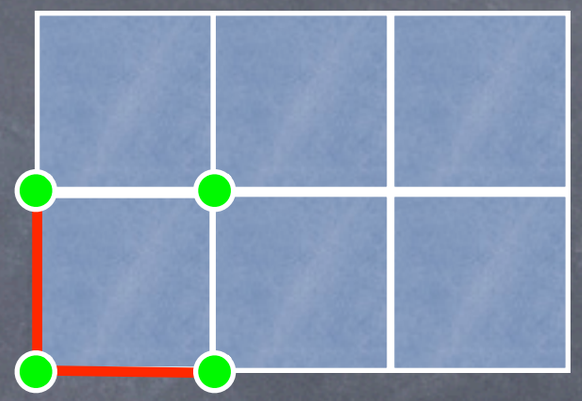
```
stream, type(real), dimension(:,:):: b(3,3)
```

```
real, dimension(:,:) :: mesh
streamSource(a,array,2,100,100)
streamGroup(b,a,STREAM_STENCIL_HALO,2,-1,1,-1,1)
```

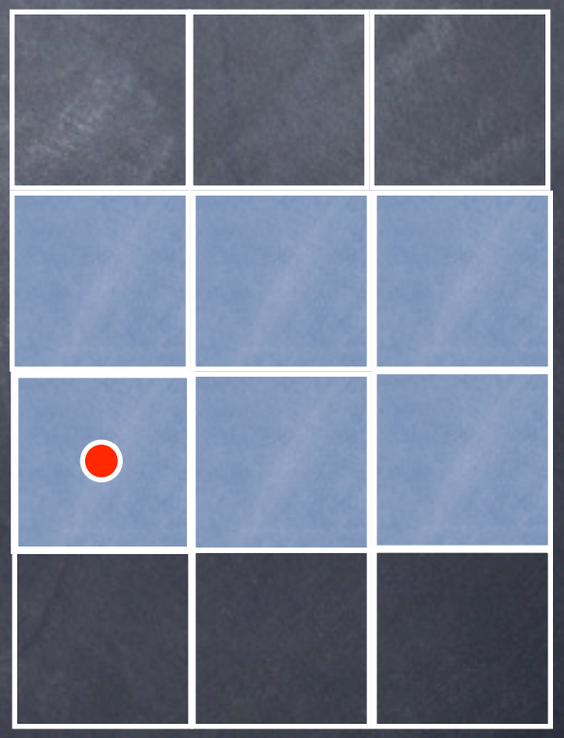
Example

```
program compute_mesh
type gridcell
            real:: x
            real:: y
end type gridcell
type(gridcell), dimension(:,:),allocatable :: mesh
real, dimension(:,:),allocatable:: vol
stream, type(gridcell), dimension(:,:):: a, b(2,2)
stream, type(real), dimension(:,:):: c
nx=3; ny=2
allocate(mesh(nx+1,ny+1),vol(nx,0:ny+1))
.....
call streamSource(a,mesh,2,nx+1,ny+1)
call streamStencil(b,a,STREAM_STENCIL_HALO,2,0,1,0,1)
call ComputeMetric(b,c,volmin,volmax)
call streamSink(c,vol(1,1),nx,ny)

kernel subroutine ComputeMetric(grid,volume,volmin,volmax)
 stream, intent(in), type(gridcell):: grid(2,2)
 stream, intent(out),type(real)::volume
real, intent(reduce):: volmin,volmax
 volume=.5*((grid(2,2).x-grid(1,1).x)*(grid(1,2).y-grid(2,1).y) &
          &   -(grid(2,2).y-grid(1,1).y)*(grid(1,2).x-grid(2,1).x));
volmin=min(volmin,volume)
vomax=max(volmax,volume)
end subroutine ComputeMetric
```

mesh



vol



Stream manipulation

- Stream load/store, domain, etc is done with function calls:

```
call streamSource(Y,X,100*200)
```

or

```
Y=streamSource(X,100*200)
```

- In Brooktran, streams have an associated shape. We should modify the load operator

```
call streamSource(Y,X,2,100,200)
```

- We can use the Fortran 9x array syntax:

```
call streamSource(Y,X(51:100,101:200),50*100)
```

Conclusions

Conclusions

- Cost/Performance: 100:1 compared to clusters.
- Programmable: applicable to large class of scientific applications.
- Porting and developing new code made easier: stream language, support of legacy codes.

- Arithmetic intensity is sufficient. Bandwidth is not going to be the limiting factor in these applications. Computation can be naturally organized in a streaming fashion.
- The interaction between the application developers and the language development group has helped insured that Brook can be used to code real scientific applications.
- Architecture has been refined in the process of evaluating these applications.
- Implementation is much easier than MPI. Brook hides most of the parallelization complexity from the user. The code is very clean and easy to understand. The streaming versions of these applications are in the range of 1000-5000 lines of code.

Plan

- Current effort is technology development
  - Demonstrate feasibility of stream technology
  - Reduce risk, solve key technical issues
- Next step is full scale development

FY02	Demonstrate simple 2D applications on single-node stream processor simulator
FY03	Demonstrate more complex 3D applications on multi-node stream processor simulator
FY04	Detailed microarchitecture, refine programming tools
FY05	Detailed design of streaming supercomputer
FY06	Construct prototype streaming supercomputers

For additional information

- http://merrimac.stanford.edu
- http://cits.stanford.edu
  - (Chapter 5, technical report)