Analysis and Performance Results of a Molecular Modeling Application on Merrimac

Erez, et al. Stanford University 2004

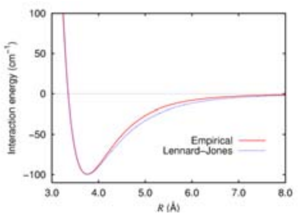
Presented By: Daniel Killebrew

What is Merrimac?

- A supercomputer composed of stream processors connected on a high radix network (fat tree)
- Single Merrimac core clocked at 1 GHz, with 2 GB DRAM costing $1000
- Scalable up to 16,384 cores

Stream Processor?

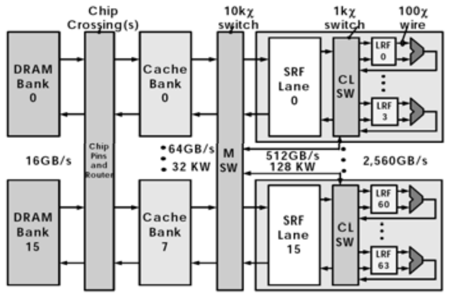
•Streaming application has a large amount of data parallelism
•Stream processor takes advantage with large number of functional units
•Backed up by deep register hierarchy
  •Local register files – locality within function call
  •Longer term locality in large stream register file



Molecular Dynamics

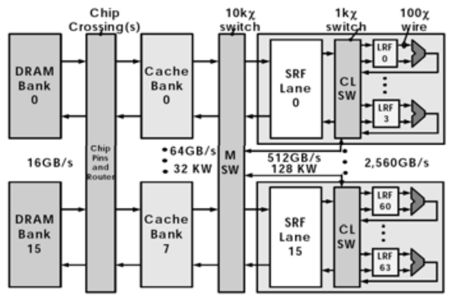
- Models (in)organic molecular systems, including protein folding and carbon nanotubes
- Calculate Lennard-Jones & electrostatic forces within a cutoff distance
- Should get performance scaling linearly with processing
- Differing number of neighbors complicates the problem, making similar to other scientific applications such as unstructured grids, sparse matrix calculations

Merrimac Architecture: local register files (LRFs)



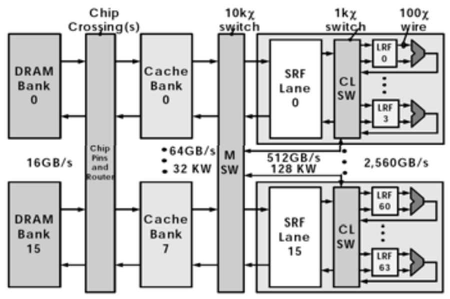
64 FPUs, organized into clusters of 4 FPUs each. 768 words of LRF per cluster (192/FPU). FPU can read 3 words/cycle and write 1 word. All clusters execute identical VLIW instruction.

Merrimac Architecture: stream register files (SRFs)



SRF capacity of 8Kwords per cluster, bandwidth of 4 words/cycle. Hides long memory latency, and jitter of memory system & interconnect.

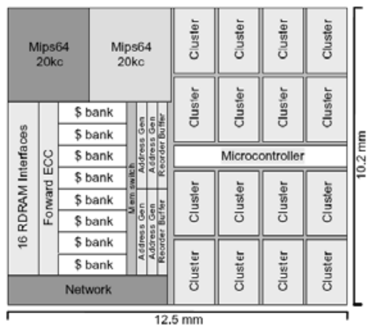
Merrimac Architecture: cache and DRAM



Cache capacity of 128Kwords, bandwidth of 8 words/cycle. External DRAM of 2GB, 2 words/cycle. Address generators support strided or scatter/gather patterns, and transfers are 1000s of words at a time from memory to SRF.

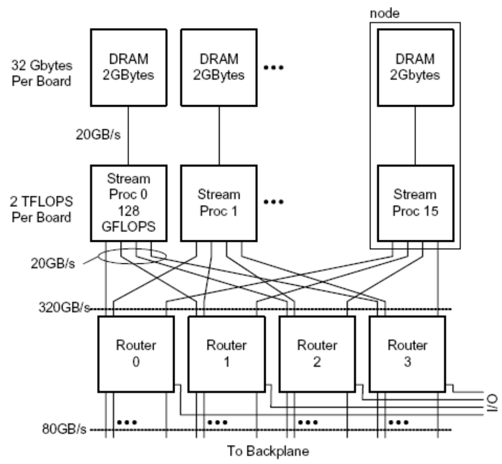
Floorplan

- 2 MIPS64 processors, one for backup
- Estimated $200 of physical mfg. costs
- 25 W of power
- 128 GFLOPS (1 madd/FPU/cycle)
- Not actually implemented…

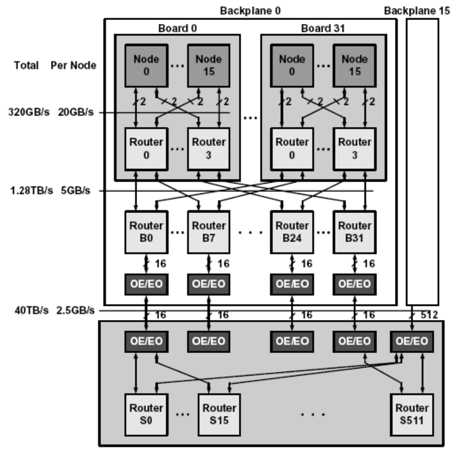


Interconnect Configuration

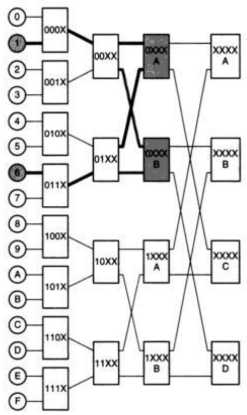
•16 cores on a board, connected to 4 routers
•Each processor has two 2.5GB/s channels to each router.
•Each router has 8 channels to the back-plane
•Back-plane consists of 32 routers connected to the 32 boards
•Back-plane optically connected to system-switch of 512 routers which joins 16 back-planes



Network Configuration



Merrimac network



Example Fat Tree

Stream Programming Model

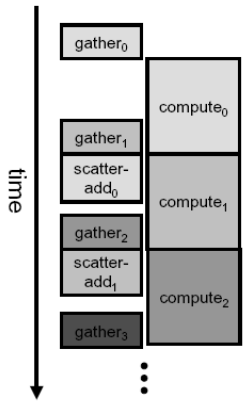
- A collection of streams, where each stream is a sequence of records, identical in structure
- A series of kernels is applied, independently to the elements of a stream
- Kernel provides the short-term locality for the LRFs
- Transfer of data from one kernel to the next provides longer-term locality for SRFs

Basic Psuedocode

- Gather interacting molecules into SRF
- Run force kernel over all interacting molecules, result is:
  - Partial forces
  - Indices mapping forces to molecules
- Scatter-add to combine partial forces into complete force on each molecule

Hiding Latency

The goal is to overlap current computation with writing the results of the previous computation, and reading the inputs of the next computation. If the SRF is large enough, should be okay.



Balancing computation and memory access

- Arithmetic intensity: the ratio of arithmetic operations performed to the number of memory operations required
- Various tradeoffs to increase the arithmetic intensity (but not all of operations performed are useful)

Implementation: expanded

- Naïve way to do it
- Read in the full interaction lists
- 2 molecules are read in, 2 partial forces are the result
- Combine partial forces into complete forces using scatter-add
- Arithmetic intensity of 4.9

Implementation: fixed

- Assume each ‘central’ molecule interacts with L other molecules
- Now we read a new central molecule every L iterations, and the neighbor at every iteration
- Reduce the forces and write the result
- At an L of 8, we reduced to 22.6 words per iteration, down from 48 of previous implementation
- Arithmetic intensity of 8.6
- Cons:
  - we must pad the neighbor list with dummy molecules when it is not a multiple of L, wasting computation/bandwidth
  - Still reading central molecules more than once

Implementation: variable

- Use conditional streams: issue the read new central molecule instruction, but execute it conditionally (same for write force)
- Don’t read in a new central molecule unless necessary (the read instruction is ignored). Likewise, don’t write out the force until all partials have been computed and reduced.
- These unexecuted instructions are wasted, but since they are not floating point instructions, kernel efficiency stays ok
- Arithmetic intensity of 9.7

Latency, revisited

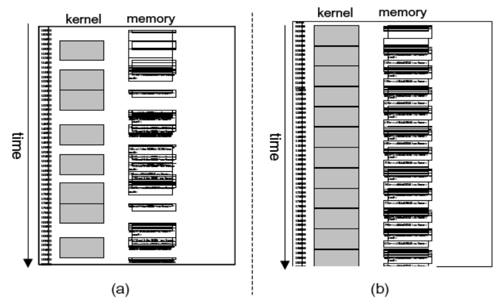


Figure 7: Snippet of execution of "*duplicated*" (see Table 3) showing the improvement in overlap of memory and kernel operations. Left column of both (a) and (b) represents a kernel being executed, and the right columns are for memory operations.

They fixed a bug and things improved…

More pictures

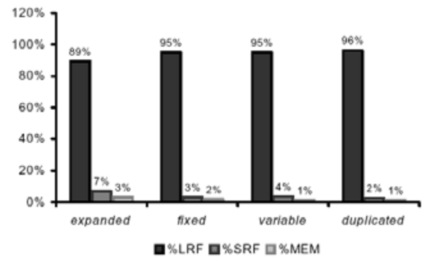
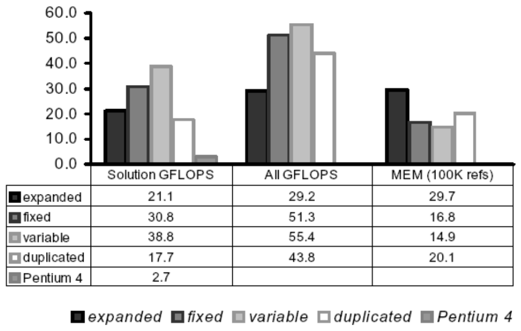


Figure 8: Locality of the various implementations

As you can see, more intelligent algorithms had increased locality due to less wasted space in the LRF/SRF/whatever.

And more



	Solution GFLOPS	All GFLOPS	MEM (100K refs)
■ expanded	21.1	29.2	29.7
▨ fixed	30.8	51.3	16.8
▨ variable	38.8	55.4	14.9
□ duplicated	17.7	43.8	20.1
▨ Pentium 4	2.7		

The left group is GFLOPS doing useful work. Central group is GFLOPS. As you can see when you compare fixed to variable, a larger number of fixed's FLOPs are not useful. The right group is number of memory references. Expanded clearly has many more references, as expected.

The Big Picture/Future Work

- Merrimac's simple code performs much better than the highly optimized P4 GROMACS code, on equivalent(??) hardware. Merrimac should be more power efficient, at least (25W vs 230W).
- Good for more than molecular dynamics. Similar problems in finite-element methods, sparse linear algebra, graph algorithms, adaptive mesh stuff.
- More algorithmic improvements: blocking (like with matrix multiplication)
- Could increase accuracy and increase arithmetic complexity at the same time (win win!).