AIAA 2004–1090

StreamFLO: an Euler solver for streaming architectures

Massimiliano Fatica , Antony Jameson and Juan J. Alonso

Stanford University
Stanford, CA 94305, U.S.A.

42nd AIAA Aerospace Sciences Meeting and Exhibit
January 5–8, 2004/Reno, NV

StreamFLO: an Euler solver for streaming architectures

Massimiliano Fatica*, Antony Jameson†and Juan J. Alonso‡

Stanford University
Stanford, CA 94305, U.S.A.

The Computer Systems Laboratory at Stanford University in collaboration with the Center for Integrated Turbulence Simulation is developing "Merrimac", a new high-performance computer system, based on a streaming architecture, that could achieve an improvement of two orders of magnitude in cost/performance compared to the current generation of supercomputer based on clusters of Symmetric Multiprocessors (SMP). In parallel with the hardware development, a new stream programming language has been developed, "Brook". In order to realize the performance of a streaming supercomputer, applications need to be converted to a stream programming model. This paper presents a short discussion of the proposed streaming architecture, our experience in re-coding a compressible Euler solver in the Brook programming language, and some preliminary results of the performance of StreamFLO using a cycle-accurate simulator for the Merrimac supercomputer.

Introduction

RECENT trends in the supercomputing market have been to cluster together hundreds or thousands of commercial servers with fast interconnection networks. The resulting systems have a theoretical peak performance in the Teraflops range, but the sustained performance in real applications is far from that peak. There are several factors that contribute to this: insufficient memory bandwidth, low network performance and scalability problems to mention just a few.

Two notable exceptions to this trend are the Earth Simulator[1] (ES) and the Cray X1. The ES is a massively parallel vector supercomputer that consists of 640 processor nodes interconnected by a single-stage crossbar switch, with a peak performance of 40.95 Tflops. A global atmospheric simulation[2] was able to achieve 65% of the peak performance and other simulations were in the 30-50% range. To put these percentages in perspective, the application Salinas,[3] a structural and solid mechanics simulation engineering package, recipient of one of the 2002 Gordon Bell Awards, was able to achieve a sustained performance of 1.16 Tflops on the ASCI White system that has a peak performance of 12 Tflops: this is less than 10%. The Cray X1 is just being deployed, but the performance of early production systems looks promising. A fully configured system should reach a peak performance of 54 Tflops. One of the reasons for the good performance of these two computers is that they are custom engineered systems with exceptional memory bandwidth, interconnect performance and vector-processing capabilities.

A major drawback of massively parallel vector supercomputers is their cost: machines like the ES or the Cray X1 use custom components at all levels. Other strategies are being pursued to achieve high performance at a lower cost, for example the Virtual Vector Architecture of Blue Planet and machines using the concept of "system-on-a-chip" like Blue Gene/L.[4]

During the last few years, the Computer Systems Laboratory at Stanford University, has shown the potential of a streaming processor for signal and image processing with the Imagine chip.[5] Imagine is a programmable signal and image processor that achieves a peak performance of 20 Gflops (single-precision floating point) and sustains over 12 Gflops on key signal processing benchmarks. Using stream processors as building blocks, a new high-performance architecture could be built. The goal of the Merrimac project (a Native American word meaning "fast moving stream"), under the leadership of W. Dally and P. Hanrahan, is to achieve superior performance through the combination of stream processors, a high-performance interconnection network that efficiently provides good global bandwidth, and a new programming paradigm to exploit this architecture. The final hardware should be able to scale from a 2 Tflops workstation to a 2 Pflops machine-room size computer with up to 16K processors. Our strategy includes a close collaboration between a team of applications developers and the hardware and language groups to design the hardware specifications and the language features.

*Senior Research Engineer, Center for Integrated Turbulence Simulations, AIAA Member

†Professor, Department of Aeronautics and Astronautics, AIAA Fellow

‡Assistant Professor, Department of Aeronautics and Astronautics, AIAA Member

This paper presents results of this collaboration in its application to CFD solvers.

In the following sections, a brief overview of the hardware and programming system will be presented to familiarize the reader with this new architecture (a detailed description of the architecture can be found in the CITS Annual Technical Report[6] and in Dally et al.[7]). The final sections will present preliminary results of the performance of a 2D Euler solver on a single Merrimac node.

Streaming Supercomputer Architecture

With modern VLSI technology, arithmetic operations (FLOPS) are very inexpensive but global bandwidth is costly. In a contemporary 0.13μm CMOS process, a 64-bit arithmetic logic unit (ALU) that can operate at 1 GHz takes less than 1 mm^2 of chip area. Hundreds of these units can be placed on a single chip. While 100 GFLOPS of arithmetic performance can be realized on one chip, the challenge is in supplying them with instructions and data. Conventional general-purpose processors that rely on global structures such as large multiported register files to provide data bandwidth cannot scale to this number of ALUs. In contrast, special-purpose processors (like the latest graphic cards) have many ALUs connected by dedicated wires and buffers to provide needed data and control bandwidth. However, special-purpose solutions lack the flexibility to work effectively in a wide variety of applications. Conventional CPUs are thus unable to exploit the potential of VLSI technology for realizing 100+ GFLOPS chips.

Recent developments enable streaming architectures that efficiently convert the capabilities of emerging technologies into realized performance on scientific applications. In the stream programming model, streams of records pass through arithmetic kernels. All the operations contained in a kernel can be applied in parallel to every record of the stream. This exposes parallelism and locality in the application. Stream processors share with vector processors the ability to hide latency, amortize instruction overhead and expose data parallelism by operating on large aggregate of data. Stream processors extend the capabilities of vector processors by adding a layer to the register hierarchy that enables them to operate in record (rather than operation) order.

A stream architecture provides large numbers of arithmetic units to exploit the parallelism exposed by the stream model. More importantly it provides a register hierarchy that can exploit the *kernel locality* and *producer-consumer locality* exposed by the stream model to greatly reduce the demand on global bandwidth. While the locality exposed by the stream model can be applied to reducing bandwidth demands in a conventional architecture, a stream processor with a large number of arithmetic units and a richer register hierarchy is required to reap the full benefit.

The Merrimac supercomputer is a shared memory parallel computer with up to 16K single-chip stream processing nodes. An overall block diagram of the Merrimac system is shown in Figure 1. The architecture leverages commodity technologies, not commodity processors, to economically achieve high performance: the main memory of the system is built entirely out of commodity high-bandwidth memory chips; the streaming processor chips are fabricated using a standard CMOS process; the system interconnect is constructed using off-the-shelf connectors and backplane technology.

The Merrimac supercomputer is designed with conservative technology assumptions. All of the technology required to build the streaming supercomputer is available today: the memory chips, the signaling technology, the connectors, the parallel optical transceivers, exist today. We expect that by the time we build this machine in 2005-6, significantly better technology will be available.

Stream Processor

Figure 2 is a block diagram of the stream processor chip that, together with 16 DRAM chips, forms a node of the Merrimac supercomputer. A stream processor consists of three independent modules:

- the scalar processor, a standard 64-bit RISC processor core, whose instruction set has been extended to include non-blocking stream instructions and whose register set has been extended to access scalar registers and stream descriptor registers in the stream processor. The scalar processor's 64-bit address space can be configured to cover some or all of the global shared memory across all the Merrimac nodes.

- a set of 16 statically scheduled SIMD clusters of floating-point units controlled by a programmable microcontroller. Each cluster, depicted in figure 3, consists of 4 fully-pipelined 64-bit floating point multiply-add units, a lookup table that gives a starting point for the iterative computation of $1/x$, $\sqrt{x}$, and $\frac{1}{\sqrt{x}}$, and an inter-cluster communication unit. With all 16 clusters operating with perfect occupancy, the peak arithmetic performance of the stream processor is 128 GFLOPS at the target clock rate of 1 GHz. The 16 clusters are connected to each other by a full 16x16 crossbar; the inter-cluster communication unit in each cluster can send and receive one word of data over the crossbar each cycle. The data-level parallelism exposed by the stream programming model is exploited by having all 16 clusters operate on different stream elements in parallel, and instruction-level parallelism is exploited by the use of multiple functional units within each clus-

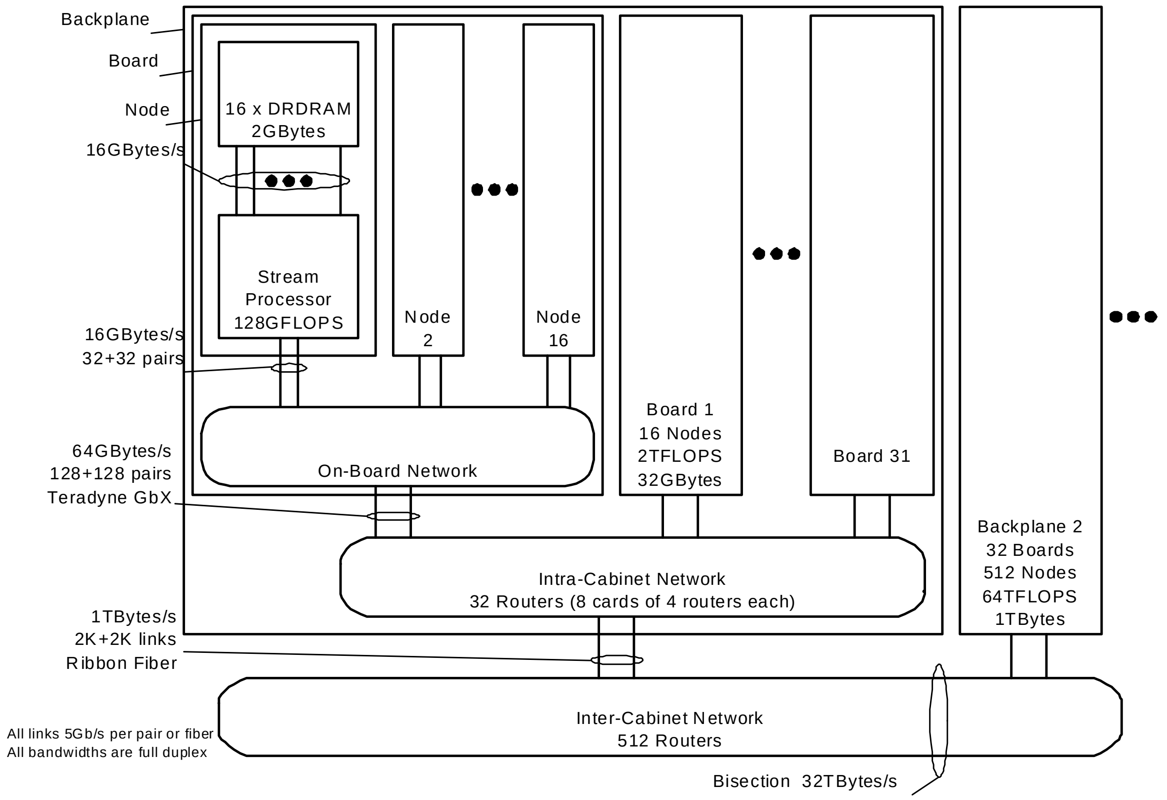

Fig. 1 Block diagram of the Merrimac supercomputing system.

ter.

- a memory system, that performs stream load and store instructions.

The scalar processor fetches instructions for all three modules from a single instruction stream and dispatches instructions to the stream processor and the memory system via a pair of instruction queues.

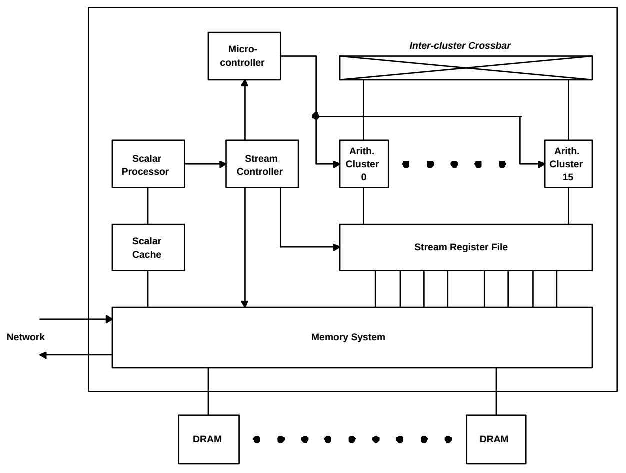

Fig. 2 Merrimac Stream Processor: block diagram

Stream instructions, which include loading and storing streams and invoking kernels, operate at the granularity of streams of data rather than individual words, and are dispatched by the scalar processor accompanied by explicit dependency information. The execution of the stream instructions is handled by the stream controller, which keeps a scoreboard of pending instructions and their dependencies as well as available resources; a stream instruction is issued to the stream hardware when its dependencies are met and the resources it requires are free. Allowing a single stream instruction to specify operations on an entire stream of data makes the stream execution units memory-latency insensitive by amortizing the latency of memory access over the many (possibly thousands) of individual words accessed by a single stream load or store.

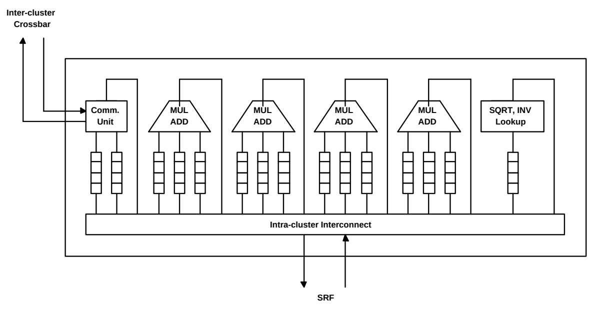

Fig. 3 Merrimac Cluster Architecture

Memory system architecture

The memory system of the Merrimac supercomputer provides high-bandwidth access to a flat address space to all of the streaming processors and scalar processors in the system. The memory system provides a cache bandwidth of 8 words per cycle (64GBytes/s) on each node and DRAM memory bandwidth to random locations of 2 words per cycle (16GBytes/s) on each node. Single word access to remote memory locations is made via the interconnection network.

Each Merrimac node has 2 GBytes of external memory distributed across 16 1Gbit DRAM chips. Depending on the final choice of DRAM chip (DDR SDRAM or Rambus DRDRAM), each node will have 38-40 GBytes/s of sequential memory bandwidth and at least 16 GBytes/s of random memory bandwidth.

The Merrimac memory system supports floating-point and integer add & store operations across multiple nodes at full cache bandwidth. Performance on add & store, which sums its argument into a memory location, is important to applications like finite-element codes in which several elements - possibly on different nodes - all sum contributions into the state of a vertex.

The bandwidth hierarchy of a Merrimac node is illustrated in Figure 4 and summarized in Table 1. At the top of the hierarchy, the 240 dual-port local register files provide almost 4 TBytes/s of bandwidth necessary to support the 128 GFLOPS of 64-bit arithmetic. These local registers directly feed the inputs and accept the outputs of the 64, 64-bit floating-point multiply-adders in each node. The bandwidth then drops by nearly an order of magnitude to 512 GBytes/s at the stream register file which holds streams of data between execution kernels. Bandwidth drops to 64 GBytes/s at the stream cache which captures temporal locality in irregular applications. The final bandwidth is 16 GBytes/s at the DRAM memory system.

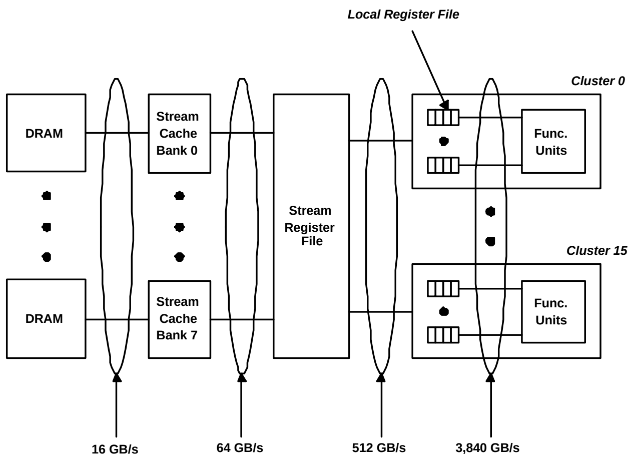

Fig. 4 Bandwidth hierarchy of a Merrimac node.

Level	Bandwidth (GBytes/s)
Local Register Files	3,840
Stream Register File	512
Stream Cache	64
DRAM Memory	16

Table 1 Bandwidth hierarchy of a Merrimac node.

Interconnection network

The interconnection network of the Merrimac supercomputer provides nearly flat memory bandwidth across up to 16K stream processing nodes. The inherent latency hiding of stream memory operations provides sufficient simultaneous outstanding memory accesses to fully exploit the bandwidth of the network.

The Merrimac network uses a folded Clos topology that enables the most efficient use of inexpensive high-bandwidth electrical signaling for short, on-board and in-cabinet, connections reserving the use of optical interconnects for the inter-cabinet routing. This topology also allows us to smoothly vary the global bandwidth of the system by varying the number of routers on the board and the degree of concentration at each router. The folded Clos topology has the further advantage that it is inherently fault tolerant. The network gracefully routes around one or more failed routers or channels.

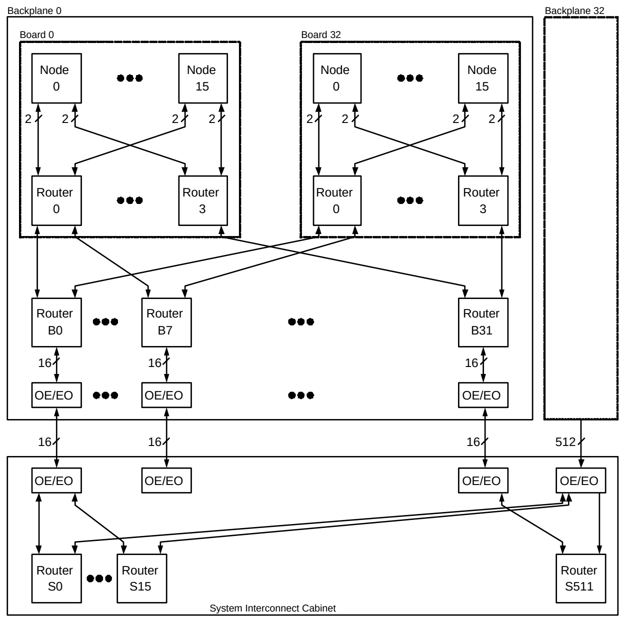

Fig. 5 The Merrimac network employs a hierarchical folded Clos topology. Each 2.5GByte/s *channel* consists of 4 5Gb/s signals. All routing is performed by a single component type, a 48-port flit-reservation router.

Brook Programming Environment

Streaming architectures achieve high performance by exploiting data parallelism and high arithmetic intensity. The programming environment therefore should expose these fundamental machine constraints in order to encourage the programmer to write code which will run efficiently on such an architecture. For this purpose, a prototype of a new streaming programming environment entitled "Brook" has been designed and built.

The current Brook programming environment extends the "C" programming language with a few new modifier keywords such as `stream` and `kernel` for specifying streaming primitives. In addition, Brook introduces a runtime library of functions which allow for creation and manipulation of streams.

A Brook program largely resembles any other C program which includes a main function, variables and function definitions. Other than the presence of these new keywords, Brook's similarity to C allows any programmer with basic C experience to understand Brook code.

Streams and Kernels

The basic programming element (or data structure) in Brook is the *stream*, a sequence of data records of the same type. Typically, streams represent the collection of data being operated on. The programmer executes functions, or *kernels*, on each element of the stream. Kernels operate on each element of an input stream and place the result of a computation on an output stream.

A Brook program consists of defining streams of data and operating on those data with a sequence of user-defined kernel functions. A critical decision in the design of Brook was to establish the constraints for kernel functions. With these restrictions in place, Brook semantics allow for efficient execution of kernels on streaming hardware by operating in parallel over the elements in a stream. Although these decisions are still not finalized, we decided to place the following restrictions on kernels:

- Global variables are not visible within kernels. Only the arguments to the kernel function are accessible.
- No data dependencies can exist between elements of a stream. No *static* variables may be declared within a kernel.
- Stream arguments are read only or write only.

These restrictions ensure data parallelism and high arithmetic intensity for Brook kernels. By removing possible dependencies between stream elements, the compiler is free to schedule the computation in parallel. Furthermore, since kernel execution is entirely local, we limit the number of global reads required for computation.

A crucial decision in the development of Brook was the support of data structures, like multidimensional arrays, commonly found in scientific applications. Streams are views of memory, sequences of references to stored data. For example, using the same array A we can generate one stream that accesses the data in column-major order and another in row-major order. Other access patterns could create block-access modes. In physical simulations, it is often necessary to access the data within a regular stencil: the local neighborhoods of a record. This neighborhood locality may be exploited in the architecture to minimize data movement.

To efficiently support multidimensional arrays in Brook, a *shape* attribute was added to streams. This shape may be used to associate a stencil around each stream element. Although this feature is common in array processing languages, it is a new feature in streaming languages, and makes coding scientific applications in Brook much easier.

Stream Operators

Brook programs consist of a sequence of kernel applications, intermixed with data movement and reorganization. Some key stream operators are:

1. `StreamLoad` and `StreamStore`. These operators create a stream from a base address and store the result of a kernel computation into main memory.
2. A stream of memory references may be created using `StreamLoadRef`. `StreamGather` and `StreamScatter` loads and stores a stream in the references contained in another stream.
3. `StreamStencil`, `StreamGroup`, These operators perform re-arrangements on streams. `StreamStencil` forms a stream of neighborhoods; the group operators combine consecutive records into a single record;
4. `StreamDomain` selects a sub-domain of a stream. For example, if the stream has a 2D shape, a 2D subset can be easily defined.

Our current goal is to settle on a small yet complete set of operators in the base system, and to complement them with a library of more complex operators built on top of those operators.

The final important feature of the language is support for reductions. One way to perform a reduction is to use `StreamScatterOp`. Another way is to declare a variable in a kernel to be a reduction variable.

The work on applications is helping the hardware group and the language developers in making decisions on the hardware specifications and on the language features.

StreamFLO

StreamFLO is a finite volume 2D Euler solver that uses a non-linear multigrid algorithm. The original FORTRAN code (FLO82) was written by Prof. Jameson and this approach is used in many industrial and research codes. The choice of the code was motivated by the need for a complete application that was representative of a typical CFD application, without unnecessary complexity.

Governing equations and discretization

The equations solved are the 2D Euler equations written in conservative form:

$$\frac{d}{dt}\iint\limits_{\Omega} \mathbf{w}\,dx\,dy + \oint\limits_{\partial\Omega} (\mathbf{f}\,dy - \mathbf{g}\,dx) = \mathbf{0},$$

where $\mathbf{w} = \{\rho, \rho u, \rho v, \rho E\}$ is the vector of conserved flow variables, and $\mathbf{f}$, $\mathbf{g}$ are the Euler flux vectors

$$\mathbf{f} = \left\{ \begin{array}{c} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uH \end{array} \right\}, \quad \mathbf{g} = \left\{ \begin{array}{c} \rho v \\ \rho uv \\ \rho v^2 + p \\ \rho vH \end{array} \right\}.$$

This set of equations is completed by an equation of state for the pressure:

$$p = (\gamma - 1)\rho \left[E - \frac{1}{2}(u^2 + v^2) \right].$$

A cell-centered finite-volume formulation is used to solve the equations together with multigrid acceleration. Time integration is performed using a five stage Runge-Kutta scheme.

$$\frac{d}{dt}(\mathbf{w}_{ij}\,V_{ij}) + \mathbf{R}(\mathbf{w}_{ij}) = \mathbf{0}.$$

On the fine mesh, a symmetric limited positive H-CUSP scheme[8,9] is used for artificial dissipation, while on the coarse meshes, a standard JST (Jameson-Schmidt-Turkel) scheme is employed.

Code organization

The code is organized as follow:

- An external driver controls both the multigrid strategy and the multigrid data movement (transfer from fine to coarse grids and vice versa). All data is stored in 1D arrays which hold the information for all the multigrid levels. Each level is selected using the *streamDomain* command and is reshaped into a 2D stream using *streamShape*. When the 3D version of StreamFLO is developed, the logical structure of this driver will not be changed, we will only need to reshape the data at each level into a 3D stream. The data movement in a multigrid algorithm consists of two operations, restriction and prolongation. The restriction operation transfers the data from a fine to a coarse mesh, while the prolongation operation does the opposite. To implement these kinds of data movement, Brook added an operator that performs local grouping.

- On each multigrid level, the code works on a 2D grid where it needs to compute the time step and the convective and dissipative fluxes in order to advance the solution in time. This involves mostly operations performed on stencils. The width of the stencil depends on the numerical scheme. Most of the operations (computation of admissible time step, convective fluxes, dissipative fluxes used on coarse meshes) are performed on a 3×3 stencil. The computation of the dissipative fluxes on the fine mesh use a more sophisticated algorithm (symmetric limited positive H-CUSP scheme) that requires a 5×5 stencil. The code uses O meshes around the surface of the airfoil being analyzed; this means that the grid is periodic in one dimension and has both a wall and an external boundary in the other direction. To handle this kind of topology, Brook was modified to implement stencil operators with independent boundary conditions in each logical direction.

Code porting

To port a Fortran/C code to Brook, there are two essential steps:

- Define the data layout and arrange the data into streams

- Define the computation kernels to be applied to the streams.

The decoupling of the data access pattern from the computation, produces a code that is cleaner and easy to understand. In order to show this and an example of Brook code, we show the code performing the restriction operator in the multigrid algorithm below. In the original Fortran code we had:

```
C TRANSFERS THE SOLUTION TO A COARSER MESH
    .........
    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)=VOL(I,J)*DW(I,J,N)
                +VOL(I+1,J)*DW(I+1,J,N)
                +VOL(I,J+1)*DW(I,J+1,N)
                +VOL(I+1,J+1)*DW(I+1,J+1,N)
       END DO
     END DO
    END DO
    .........
```

we can see that all this routine does (see figure), it is to combine the change in the flow solution (DW) in 4 cells on the fine grid (A,B,C,D), into a single cell (AA) on the coarser level. The algorithm then moves to the next 4 cells and continues until it covers all the interior of the fine domain.

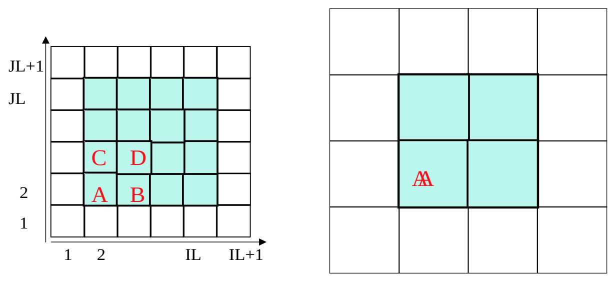

Fig. 6 Restriction operator

The first operation we need to perform is to create the right stream from the fine grid to feed the kernel. We want to take the fine and coarse grids, select the interior elements, arrange the fine grid in 2×2 groups and apply the TransferField kernel. The arrays on the fine grid are dimensioned as $(nx + 2) \times (ny + 2)$.

```
// Fine mesh:
//   fine_flow is a 2D stream of
//   shape (nx+2,ny+2)
// Coarse mesh:
//   coarse_flow is a 2D stream
//   of shape (nx/2+2,ny/2+2)

// Select the interior cells on the fine mesh
streamDomain(flow,flow,2,2,nx+1,2,ny+1);
streamDomain(vol,vol,2,2,nx+1,2,ny+1);

// Group elements of the fine grid
// in a 2x2 stencil
streamGroup(local_flow2d,flow,
            STREAM_GROUP_HALO,2,2,2);

//  Apply restriction operator
TransferField(local_flow2d, coarse_flow)
```

The following Brook code for the kernel function performing the restriction operator will sum the 4 element on the fine grid stream and store the result in the corresponding element on the coarse grid stream.

```
kernel void TransferField(flow2d_s fine_flow,
                   float2d_s vol ,
                   outfixed Flow coarse_flow)
{
float volc=(vol[0][0]+vol[0][1]
           +vol[1][0]+vol[1][1]);
for (int n=0;n<4;n++) {
  coarse_flow.w[n]=
         (
          vol[0][0]*fine_flow[0][0].w[n]
         +vol[0][1]*fine_flow[0][1].w[n]
         +vol[1][0]*fine_flow[1][0].w[n]
         +vol[1][1]*fine_flow[1][1].w[n]
         )/volc;
                            }
}
```

The code is less compact than the original Fortran code but is easier to understand and to modify.

Implementation of boundary conditions

The implementation of boundary conditions in a SIMD program is not a simple task. The numerical treatment of the elements close to the boundary is generally different from the one used for the interior points, in contrast with the SIMD philosophy of applying the same instructions to multiple data. Inside the kernels, depending on the global position of an element in the original stream, we may need to have different branches of code to handle this. In Brook, when a stream enters a kernel, all the information regarding the global position of an element is lost (at least from the programmer's point of view). The solution found to overcome this problem is to encode the information about the position in the geometry. The original definition for the mesh:

```
struct Grid_struct {
float x;
float y;};
typedef stream struct Grid_struct Grid;
```

was modified to include information about the logical position.

```
struct Grid_struct {
int i; /* logical position in x direction */
int j; /* logical position in y direction */
float x;
float y;};
typedef stream struct Grid_struct Grid;
```

In a conventional cache based machine, this modification results in a cache load of all the 4 elements of the Grid struct (they are contiguous in memory) for every reference to any element and therefore in a waste of precious cache space and bandwidth. In the streaming architecture, the compiler will analyze the use of the elements of the Grid struct and will load in the stream register file only the elements used by the kernel. If a kernel does not need information on the logical position of a cell but just its physical coordinates, it will not waste stream register file space with elements that it is not going to use. Another possible solution is to use a mask array. For some situations, it is also convenient to write a different kernel that just deals with the boundaries, but this is not always possible.

Results

We have two implementations of the Brook language: one is a compiler that translates Brook to C and then links with a set of runtime libraries for the stream manipulation. The other is a compiler that produces code for a cycle-accurate simulator based on the Imagine tools (from previous experience the results from the simulator have a margin of error of $5 - 10\%$). We use the first implementation to check correctness, perform language studies and to run on standard hardware (the compiler is based on gcc and runs on several platforms). The second tool is used to predict the performance of the code on actual hardware (when it becomes available). Both tools are still under development and the optimization is far from optimal. The cycle-accurate simulator does not yet support multiply-and-add instructions, so for these preliminary results the peak performance of the node is only 64 Gflops. Nevertheless, the results from the initial simulations on the cycle accurate simulator are very encouraging. The schedule of the kernel for the computation of the diffusive flux using a symmetric limited positive H-CUSP scheme is shown in figure 7. The simulator indicates that this kernel in its current form will run at 37 GFlops, 58% of the peak performance. Improving the set up of the stencil operator (that is now using only 20% of the resources) should bring the sustained performance up to 50 GFlops, close to 80% of peak.

According to the simulator, the full application will be able to sustain 11.4 GFlops (18% of peak), performing 7.4 floating point operations for each global memory access. Note that only real ops are counted in this figure, such as floating point add/mul/compare instructions, and not non-arithmetic ops such as branches. Divides are counted as single floating point operations, even though each divide requires several multiplication and addition operations when executed on the hardware. This leads to the lower performance numbers – for example, the sustained performance of StreamFLO would double if we counted all the multiplies and adds required for divisions as well. The hardware design group is currently looking at improving the performance of divide operations. Once again, it should be stressed the preliminary nature of these performance number: the tools are still in an early stage and far from optimal performance.

Brooktran

In order to simplify the porting of pre-existing Fortran code to Merrimac, we are developing a new variant of Brook, called Brooktran, that will be a streaming extension of Fortran.

Brook is an extension of C which allows incremental porting of pre-existing C codes: computationally intensive functions can be individually replaced with kernels to take advantage of the stream processors (a kernel is a piece of code that will be executed on the streaming processor unit). This incremental streamification, similar to the incremental parallelization possible with OpenMP, greatly simplifies porting an existing C code to Merrimac. The code will run with no changes on Merrimac, and increasing performance can be obtained by converting regular functions to kernels. For pre-existing codes written in Fortran the situation is completely different. The whole code needs to be rewritten in C and Brook, with their many differences from Fortran, making the task even harder than, say, parallelizing a serial code using MPI. Fortran is still very popular in the High Performance Computing community. The national laboratories, national agencies (such as NASA), universities, and many industries have millions of lines of existing, valuable Fortran codes. This investment in Fortran can be expected to continue with the advent of the Fortran 90 and 95 standards, which include modern memory management and data and control structures. Porting all these codes to C/Brook is not feasible, particularly considering the validation efforts that have been spent bringing the codes to their current state of acceptability. Some of these codes are used for mission critical tasks which require their results to have a very high level of reliability. The purpose of Brooktran, a streaming extension of Fortran, is to give an incremental porting path to the Fortran community. In order to have an interface consistent with the C counterpart, we will adapt certain Fortran 90 features, such as the intent attribute, to the language. This also facilitates use in a mixed-language environment. The kernels are written using a Fortran syntax and have the same constraints as Brook kernels. As in Brook, the setup of streams is done through library calls.

Future work

The project is still under active development. In particular, the Brook syntax has been refined and a new compiler infrastructure is under development. A 3D Navier-Stokes solver is being developed to further study the validity of the approach. The solver contains a large portion of the execution kernels of the three-dimensional, Reynolds Averaged Navier-Stokes flow solver TFLO (the main RANS application for our ASCI Center) and is intended to provide performance benchmarks for realistic, industrial-strength flow applications.

The authors think that major breakthroughs to the aerospace industry will come from desktop machines based on Merrimac nodes, when every engineer will have the capability of running high fidelity 3D simulation on their desks in a short amount of time. We anticipate that it should be possible to demonstrate working hardware in 2006-2007 (the actual schedule will depend on availability of funding)

Acknowledgments

This work is supported by the US Department of Energy under the ASCI program. The authors would like to express their gratitude to the other members of the Merrimac project, in particular to Mattan Erez and Ian Buck. The section on the Merrimac computer

architecture is a summary of the chapter in CITS Annual Technical Report[6] written by W. Dally and P. Hanrahan.

References

[1]Yoshida K., Shingu S.,"Research and Development of the Earth Simulator" Proceedings of the 9th ECMWF Workshop. World Scientific, 2000, pp 1-13.

[2]Shingu S., Takahara H., Fuchigami H., Yamada M., Tsuda Y., Ohfuchi W., Sasaki Y., Kobayashi K., Hagiwara T., Habata S., Yokokawa M., Itoh H. and Otsuka K., "A 26.58 Tflops Global Atmospheric Simulation with the Spectral Transform Method on the Earth Simulator", SC2002 Proceedings.

[3]Bhardwaj M., Pierson K., Reese G., Walsh T., Day D., Alvin K., Peery J., Farhat C. and Lesoinne M., "Salinas: A Scalable Software for High-Performance Structural and Solid Mechanics Simulation" , SC2002 Proceedings.

[4]McCurdy W., Stevens R., Simon H., Kramer W., Bailey D., Johnston W., Catlett C., Lusk R., Morgan T , Meza J., Banda M., Leighton J. and Hules J., "Creating Science-Driven Computer Architecture: A New Path to Scientific Leadership", LBNL, October 2002,Publication 5483.

[5]Khailany B., Dally W. , Rixner S. , Kapasi U., Mattson P., Namkoong J., Owens J., Towles B., Chang. A., "Imagine: Media Processing with Streams". IEEE Micro, Volume: 21, Issue: 2, March 2001.

[6]"2002 Annual Technical Report", Center for Integrated Turbulence Simulations, Stanford University, edited by M. Fatica, W.C. Reynolds and J.J. Alonso, *available at http://cits.stanford.edu.*

[7]William J. Dally, Patrick Hanrahan, Mattan Erez, Timothy J. Knight, Franois Labont, Jung-Ho Ahn, Nuwan Jayasena, Ujval J. Kapasi, Abhishek Das, Jayanth Gummaraju and Ian Buck, "Merrimac: Supercomputing with Streams" ,SC2003, November 2003, Phoenix, Arizona.

[8]Jameson, A.,"Analysis and design of numerical schemes for gas dynamics 1. Artificial diffusion, upwind biasing, limiters and their effects on accuracy and multigrid convergence" International Journal of Computational Fluid Dynamics, Volume 4, 1995, pp 171-218.

[9]Jameson, A.,"Analysis and design of numerical schemes for gas dynamics 2. Artificial diffusion and discrete shock structure" International Journal of Computational Fluid Dynamics, Volume 5, 1995, pp 1-38.

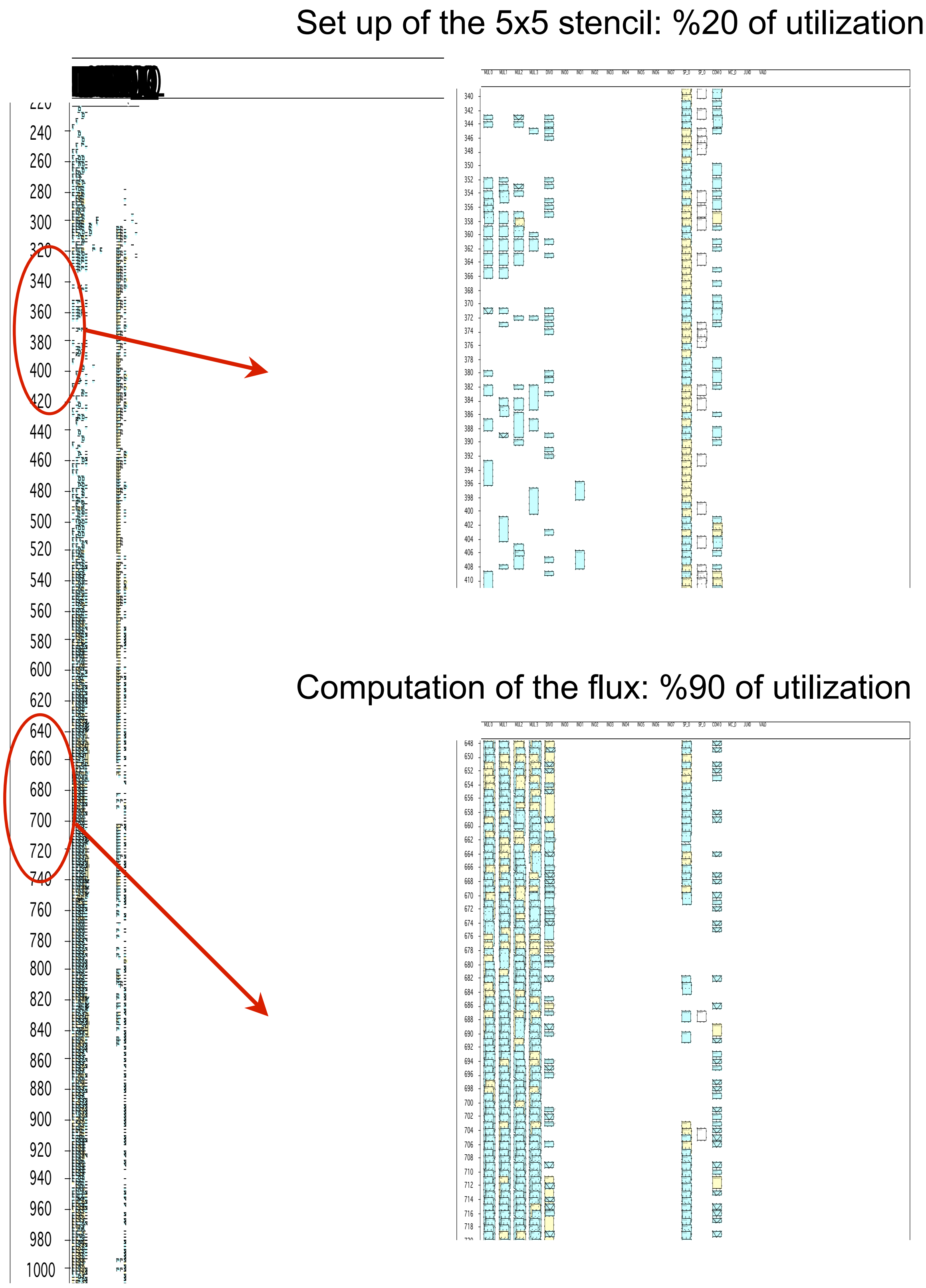

Fig. 7 Kernel schedule for the H-CUSP dissipative flux calculation: Clock cycles are displayed in the vertical column on the left and the next columns display the utilization of the various resources of the SIMD cluster. Magnification of the encircled areas are shown on the right. The 4 left-most columns in the magnified section are the 4 multiply and add units, the fifth column represents the lookup table for the inverse and square-root operations. The 3 right-most columns are related to the intracluster communications. Filled spaces indicates full utilization, while empty spaces indicate wasted resources.