Improving Unstructured Mesh Application Performance on Stream Architectures

Nuwan Jayasena, Francois Labonte, Yangjin Oh,
Hsiao-Heng Lee, and Anand Ramalingam

Stanford University

6/6/02

1 Summary and Findings

In conventional stream applications, data is usually organized as streams of regular records. As a result, SIMD microprocessors such as the Stanford Imagine Processor are well suited for regular streams, since arithmetic clusters can process records of identical size in a parallel lock-step fashion. For applications where records have variable sizes, the lock-step execution of the SIMD architecture can hinder performance, since the throughput becomes limited by the longest record size in each record fetched.

One application class with irregular data structures is unstructured mesh computation. In addition to having an irregular record size due to a varying number of neighbors per node, each node is visited several times in the course of computation by its neighbors, thereby increasing the possibility of data reuse. In this project, we propose different approaches to enhance the current Imagine Processor architecture in order to support unstructured mesh applications.

First, we make the general data structure of the unstructured mesh regular by separating the "data" information from the "neighbor" information in order to better support computation kernels. Motivated by the possibility of data reuse, we then examine the performance enhancement by adding a cache in the Imagine memory system, and conclude that only a small cache is needed to achieve a 4x performance improvement for memory-limited applications. Another technique we explore to exploit repeated access to data is indexing into the stream register file (SRF). Because each node element is only stored once in the SRF, we can effectively increase the data strip size and reduce memory traffic to increase performance. Software hashing and hardware arbitration are used to mitigate bank conflicts in SRF indexing. Simulation results indicate that SRF indexing can also reduce execution time by 50%-75% for memory-limited applications with little hardware overhead.

2 Unstructured Mesh Applications

Unstructured mesh applications are common in scientific domains such as computational fluid dynamics, thermodynamics and others. The dataset is composed of a set of vertices connected to some neighbors. The degree of connectivity varies for each vertex. Usually the simulations update values of a vertex based on the values of neighbors. The unstructured mesh in an application can be static with no topology change or adaptive based on the application need for refinements or coarsening.

Traditional computing architectures have had to tackle the issues of partitioning the mesh across processors with the least amount of overlap between partitions, balancing the load on each processor and exploiting the data locality at the higher level of the memory hierarchy. Our project tries to address the issues of load balancing and memory optimizations for unstructured mesh applications on stream architectures. The data partitioning between stream processors is no different than for conventional processors and doesn't require special attention.

The data level parallelism inherent to stream processors like Imagine through SIMD clusters requires special attention for load balancing. Our first goal will be to find proper techniques to load-balance an unstructured mesh application on stream architectures.

The memory system of Imagine is one where the data streams consumed by the clusters are loaded in the stream register file before the kernel can execute. This allows no memory stalls during kernel execution and a predictable access pattern of the stream that allows us to maximize the SRF bandwidth. This model doesn't map efficiently for unstructured mesh applications.

Because of the irregularity of neighbor connectivity, the data accesses are not regular and need to be made regular in the memory. The basic idea is that for each element to be updated we determine the neighbors required, and we load neighbors for that vertex in memory, then process the elements. The 2 problems that are encountered is that the number of neighbors differ and that even if there is a lot of vertices which are the same for different vertices, they each take up space in the SRF. Besides the neighbor copies need to be loaded multiple times from the memory system which amplifies the demands on the memory system artificially.

To tackle the issues of the memory system, replication and useless demands on the memory system, we propose two architectural changes to the current imagine stream architecture, a cache between the SRF and the memory system and a way to index into the SRF.

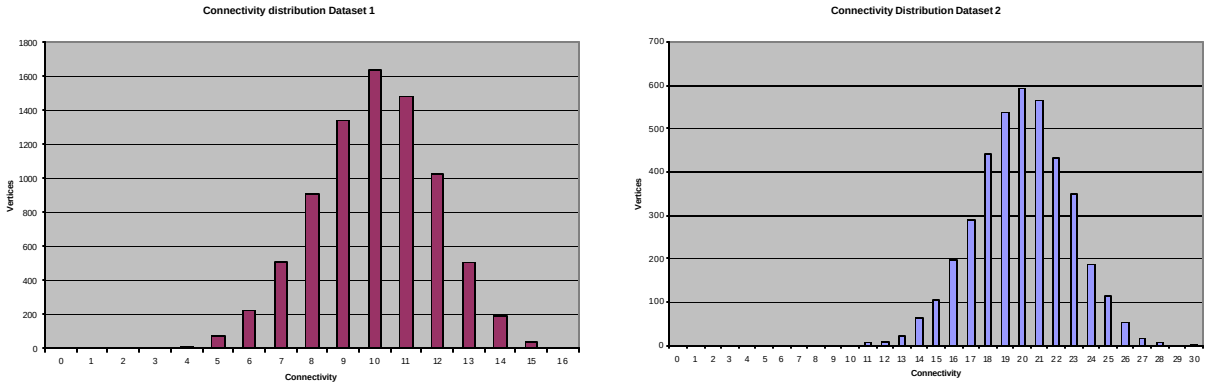
3 Unstructured Mesh on Imagine

To address the issue of load balancing unstructured mesh applications on a stream architecture across SIMD clusters we coded an unstructured mesh application in different

ways in KernelC and StreamC for Imagine. Using ISIM, the cycle accurate simulator for Imagine it will also be possible to observe behavior of the current Imagine configuration and one with a cache and performance improvements.

The replication and locality of the data set will affect the performance improvement possible by including a cache or by allowing indexing into the SRF.

The histogram of connectivity of both data sets is in the following figure.



It is essential to have regular data structures to be able to regularize the data access for our stream application. The basic data structure chosen is simple with a fixed record size. Each vertex record contains its values and a pointer into the connectivity array and the number of neighbors it has. The connectivity array contains indices into the vertex array of connectivity.

3.1 Algorithms

The most important aspect of the kernels is how the neighbors will be accessed. On current Imagine, the clusters first need to produce a stream of the indices of the required neighbors. The next important aspect is how to handle the disparity of the number of neighbors, which will have impacts on the way the SRF is packed and on the load balancing.

To generate indices into the vertex stream, the smart way to do it, is to look only at the vertex's pointer into the connectivity stream and their number of neighbors. The idea is to generate first indices into the connectivity stream, index load it and then index load the vertices themselves. For a static mesh like the ones we are considering, the process of generating indices into the connectivity stream only needs to be done once, because the resulting stream of indices into the vertex stream can be saved back into memory for

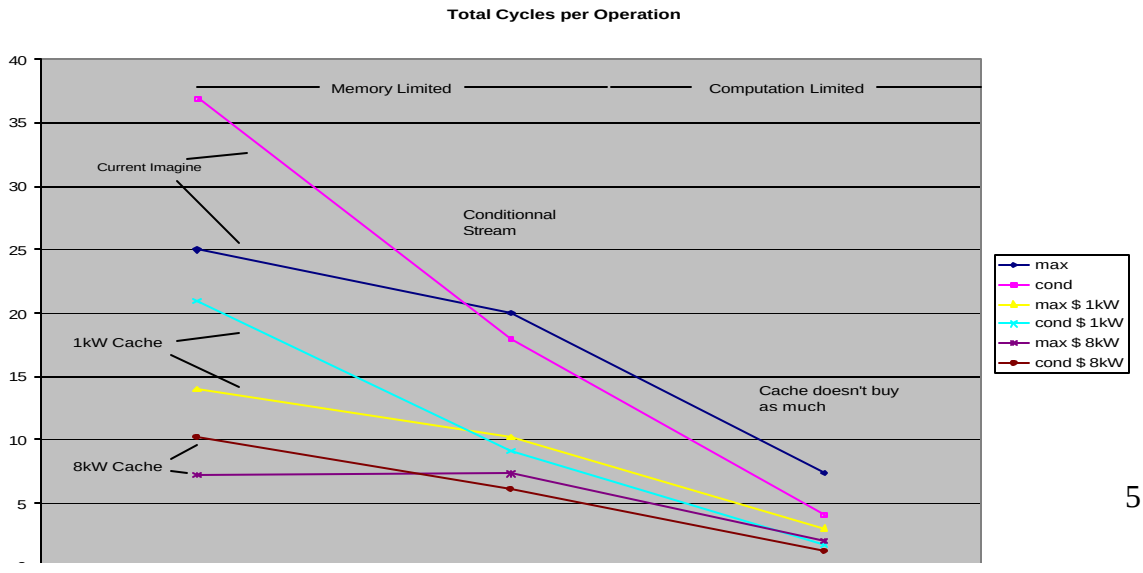
subsequent iterations. So a cluster generates indices into the connectivity stream starting at the pointer and generates increments of that pointer up to its number of neighbors. An easy way to do load balancing is to generate the maximum number of possible neighbors for each vertex, generating a neutral vertex's index when there are no more neighbors. This is wasteful of the SRF space, and it puts an even bigger load on the memory system by having to load the neutral element extremely often.

Another way is to use conditional streams to start operating on the next vertex once you are done with the current one. In the kernel that fetches the streams, similarly, once the neighbors of one vertex have been fetched, the neighbors of the next vertex are fetched. The level of granularity at which this should be done depends on the number of operations to be made on each neighbor and the variance in the number of neighbors of the dataset. To measure extreme cases this was done at the granularity level of one neighbor.

On a stream architecture using double buffering, the memory operations and kernels operate at the same time, and an application will be limited by the longest of these two operations. The number of operations per neighbor will help determine if it is limited by memory or cluster computation. To see this effect in our study, we have developed kernels of both kinds (max and cond) which have respectively 1, 10 and 100 operations per neighbor.

3.2 Simulations and Results

The two types of kernels, max and cond have been simulated on Imagine's cycle-accurate simulator ISIM with kernels of 1, 10 and 100 operations per neighbor. The application as



strip-mined and double-buffered to be able to see when it would be memory or compute limited. Also of interest are the improvements that a cache provides. Since a cache can be included in ISIM as it is, we will make use of it. To see the impact of the cache size on the performance, the same kernels were simulated with a cache of 1k words and 8k words.

The graph above gives the results for the simulations. It is obvious that the application is memory limited for both the 1 and 10 operations per neighbor kernels. Cache buys a lot more in these circumstances. The max kernel even with its shorter strip size is faster than the cond one at low operations per neighbors, because of the overhead of conditional streams. Above 10 operations per neighbor, the conditional stream approach is faster because of the bigger strip size and the amortizing of the conditional stream on more operations.

The cache improves performance a lot when the application is completely memory limited, up to 4 times for 1 op per neighbor, max kernel. When the kernel reaches 100 ops per neighbor, the application is compute limited and the cache doesn't buy much. It seems that the point at which the application becomes compute limited occurs between 10 operations per neighbor and 100 operations per neighbor. This is consistent with the ratios of arithmetic intensity to memory bandwidth of Imagine, at around 15 ops per memory reference.

In terms of the best cache size it is obviously depend on the connectivity of the graph, a graph with a high degree of connectivity will have a lot of redundancy and could be accommodated with a smaller cache. In our case of graphs with an average of 20 and 10 neighbors, a cache of 8k words seems to have bought us much better performance than a 1k words cache for kernel of low arithmetic intensity.

4 Indexing in to SRF

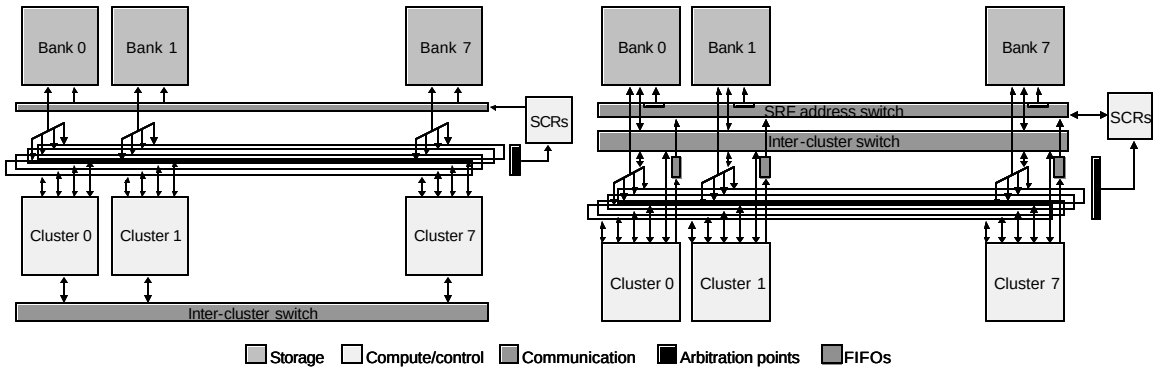
The general idea of SRF indexing is to allow clusters access to arbitrary locations in the SRF by providing explicit addresses. This is in addition to, and not a substitute for, sequential streams. The primary benefit of indexing for irregular mesh applications is a reduction in data replication in the SRF. For any graph with average degree *N*, a linear stream of the neighbors of all nodes contains, on average, *N* copies of each node record. Therefore, for reasonably dense graphs (*N* >

1), there is replication in the neighbor stream. SRF indexing, by removing the sequential access constraint, allows repeated access to a single copy of each record in the SRF.

The entire data set will not fit in the SRF for most realistic meshes. Once stripmined, the average data reuse in the neighbor stream can be expressed as $R = N$. The difference between R and N is due to replication at strip boundaries. However, this overhead is fairly small for graphs with locality since indexing allows larger strip sizes by eliminating replication within a strip. Another direct impact of reducing data replication in the SRF is the reduction of bandwidth demands on the memory system for stream transfers.

4.1 Implementation Model

A conceptual view of a stream processor (Imagine) is shown in figure A(a), and the proposed implementation of SRF indexing on the same processor is shown in figure A(b). Primary modifications include using the inter-cluster network to facilitate indexed accesses from any cluster to go to any SRF bank, and adding mechanisms to allow clusters to issue explicit SRF addresses. The indexed accesses are treated as another stream when arbitrating for the SRF bandwidth. Explicit addressing is done by placing SRF indices in to address FIFOs associated the clusters. In the base implementation, each cluster has a dedicated address bus (14 bits each for the 16K-word Imagine SRF) used to broadcast the head entry of its FIFO to all SRF banks. Each bank performs arbitration among potential multiple requestors every cycle independently. When an access is completed, the data is routed back to the requesting cluster over the inter-cluster network and placed in a stream buffer. Several enhancements to this base architecture will be presented in the results and discussion section. We also explored a more advanced scheme where addresses from the address FIFOs are issued out of order, and the data reads are re-ordered before returning to the cluster in issue order. A cost/performance analysis of this model will also be presented later in this report.



(a) Original stream processor (b) Stream processor with SRF indexing

Figure A: Hardware organization

In order to tolerate the long and unpredictable SRF access latency, the address issue and data consumption for indexed accesses is decoupled from the clusters' perspective into two separate operations. All clusters stall when any cluster fails to issue an address due to a full FIFO or fails to read due to an empty stream buffer. In order to avoid deadlock, the number of outstanding addresses before the corresponding data reads are issued is limited to the number of FIFO entries, and this requirement must be satisfied by the compiler. In order to simplify programming and to allow the compiler freedom to enforce the above constraint, it is assumed that the programmer views indexed accesses as single operations (i.e. not decoupled).

Our current experiments only use indexed reads. Supporting indexed writes is fairly straightforward, but the hardware will not provide any ordering guarantees if multiple clusters write to the same SRF location (similar to ordering issues in other parallel systems). We maintain the requirement that all data in the SRF (including indexed data) be either read only or write only.

The primary drawback of indexed accesses compared to sequential streams is the inability to utilize wide reads out of SRF banks since indexing at word granularity inherently requires column multiplexing at the SRAM array outputs. In the case of Imagine, this results in a 4:1 reduction in bandwidth for indexed accesses. However, this is more than compensated by the reduction in memory traffic for kernels with low compute intensity. Kernels with high compute intensity are insensitive to this since compute resources typically limit their performance.

4.2 Application Mapping

Structuring irregular grid computation to make use of SRF indexing requires mapping node IDs to an index in to the SRF. Our base implementation uses an application-independent hash to determine the SRF bank to place each neighbor record of a strip. Once the bank is determined, the record is appended to the records list in that bank if it is not already there, and the resulting *<node ID, bank, index in bank>* mapping is added to a hash table. Note that there is one hash table per strip. The neighbor pointers of node records are then modified to use *<bank, index in bank>* references instead of absolute node IDs. This set of tasks is not compute intensive and needs to be performed only during mesh repartitioning (incremental mesh adjustments can be accommodated with incremental changes). Therefore, we currently assume this is done on the host processor.

The main compute kernel takes inputs of a stream of nodes, a table of neighbor records, and a stream of pointers to the neighbor table. The stream of pointers specifies the linear sequence of neighbor references for the stream of nodes. Note that if the data fields required in the node

records are the same as those required in the neighbor records, the node stream can also be replaced with an index stream in to the table. However, we are not assuming this optimization.

4.3 Tools Developed for SRF Indexing Evaluation

Application-specific statistics and trace generator. This tool can be used to gather statistics on distribution of records to SRF banks, data reuse and bank conflicts in indexed SRF accesses, strip sizes, etc. for various hash functions and hardware parameters. Also generates SRF access traces for simulation.

Trace-driven SRF simulator. This is an application-independent simulator that accurately models SRF accesses, access conflicts, and cluster network conflicts for conditional and unconditional sequential streams, indexed accesses, and memory system streams. The memory system itself is not modeled, but an ideal system is assumed (this is the worst case for determining the interference to cluster SRF accesses due to memory streams). The Imagine memory system parameters were used for our studies.

4.4 Simulation Results

In order to study the application-level impact of SRF indexing, we analyzed the impact on strip size and the total memory bandwidth consumed for one iteration of the entire mesh. Figure B shows the strip sizes for a mesh with average node degree 10 that fit within 8K-words of SRF space. This corresponds to a 2-stage *Stream C*-level software pipeline in the 16K-word Imagine SRF. For the record sizes explored, indexing allows 1.5x to 3.5x larger strip sizes compared to sequential streams. Increasing the node degree shifts both curves down, but the curve for the indexed case shifts down at a much slower rate since only the index stream size grows.

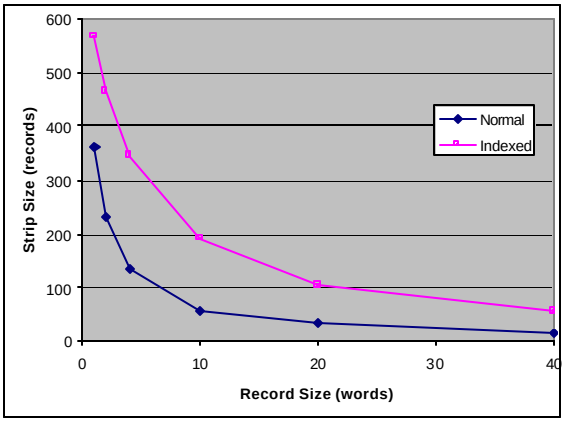


Figure B: Maximum strip size vs. record size

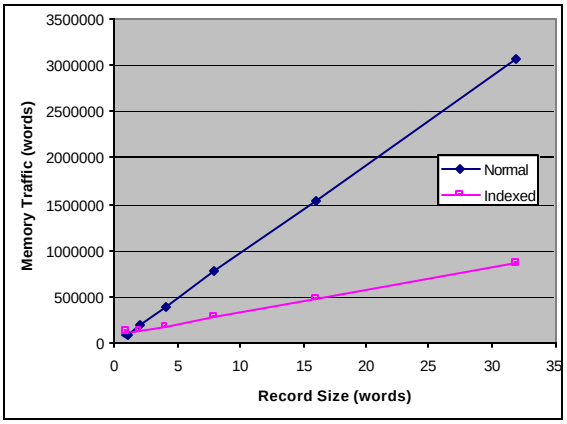
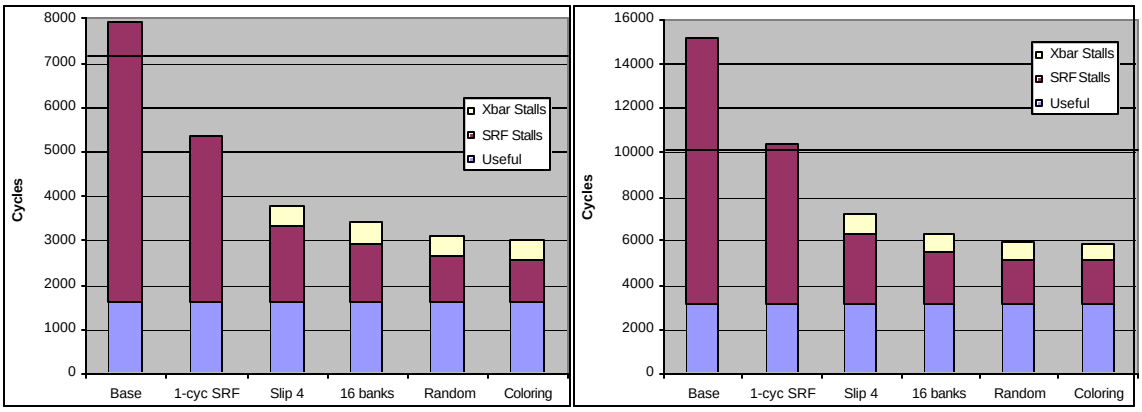


Figure C: Total memory traffic vs. record size

(8K-words SRF space, avg. node degree 10) (avg. node degree 10, strip size 256)

Figure C shows the variation in total memory traffic for the indexed and non-indexed cases as the record size varies from 1 to 32 words for a mesh with average node degree 10. The indexed method generates slightly more (about 17%) traffic when record size is 1 due to the neighbor table since the SRF index stream requires as much space as the neighbor stream of the non-indexed case. As the record size increases from 2 to 32 however, the memory traffic reduction due to indexing increases from 27% to 72%. Again, higher node degree would degrade performance for normal streams at a much faster rate than the indexed case. There is also a slight improvement in total traffic for larger strip sizes in the indexed case due to reduced replication at strip boundaries. This overhead for the above mesh reduces from 50% to 3% as the strip size increases from 16 to 256.

To study the impact of indexing on kernel execution time, we simulated the SRF accesses for the interaction kernel using traces. Zero compute time and an ideal memory system were assumed in order to force the SRF to be the limiting resource. Figures D(a) and D(b) show the execution time for one invocation of the kernel for two graphs of average degree 10 and 20. Record sizes of 4 words were assumed for both.



(a) Mesh of avg. degree 10 (b) Mesh of avg. degree 20

Figure D: Execution time for various optimizations

The *base* configuration is essentially the current Imagine architecture with minimal indexing hardware – essentially just the address busses, 1-entry address FIFOs, and inter-cluster network connections to the SRF and stream buffers. The subsequent bars correspond to incremental enhancements. The *base* case has a large number of stall cycles due to inadequate SRF bandwidth. The *1-cyc SRF* case improves on this by pipelining the SRF to issue one access every

processor cycle (as opposed to every 2 processor cycles in the current design), which we believe can be achieved with fairly low hardware cost. The *slip 4* case adds 4-deep address FIFOs allowing some "slip" between the address sequences from different clusters. Note that by this point, contention in the inter-cluster network between indexed and conditional reads starts to limit performance as well. *16 banks* case doubles the number of SRF banks. Since larger SRAMs can be built efficiently using smaller SRAM arrays (in fact, this becomes the more economical option for larger SRFs) we believe this to also be a fairly low overhead modification. Note that the number of clusters is unchanged, and therefore the impact on the inter-cluster network area should be minimal. The *random* case randomizes the neighbor access order, slightly reducing the likelihood of many nodes requesting the same neighbor simultaneously. The *coloring* case replaces the application-independent hash with a local, greedy, pseudo-graph-coloring approach to allocate records to banks. This provides a small improvement, but requires preprocessing in order to analyze the data set. Note that the stall overhead for the last few cases is less than 50%, implying that 2 cycles worth of computation per word of data read is adequate to make the application compute bound.

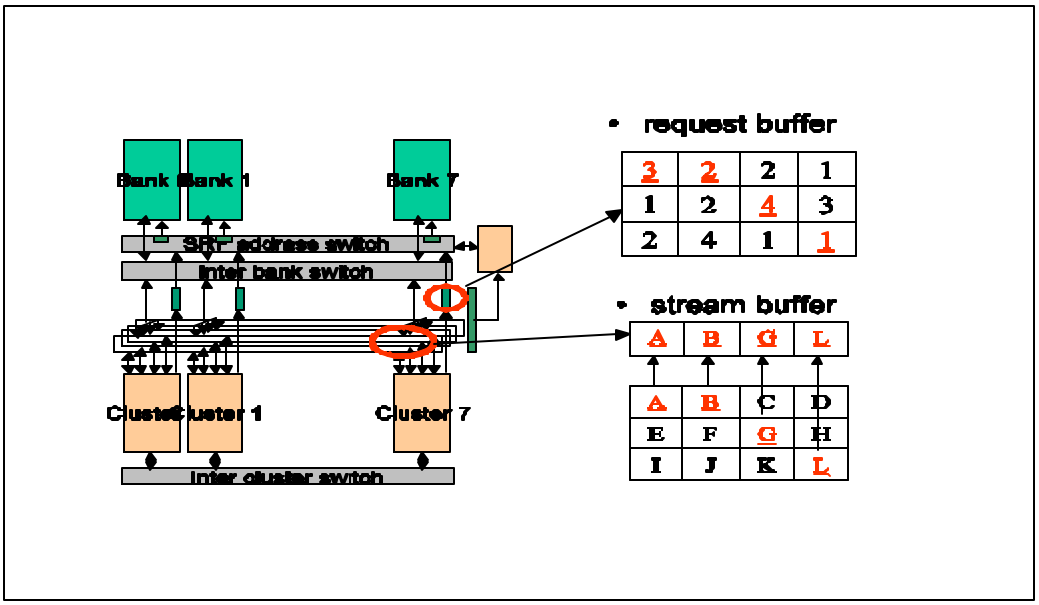
The thick horizontal line on each of the plots in figure D indicate the point at which the application becomes memory-bound even with the reduced memory bandwidth needs due to indexing and assuming a perfect memory system. This shows that SRF indexing with a few low cost optimizations can improve performance of memory bound applications without creating a new bottleneck. Even with no optimizations, the memory and compute are fairly well balanced even in the worst-case scenario of zero computation presented above.

5 Reducing Bank Conflicts with Hardware Arbitration

In addition to the "base" SRF indexing scheme discussed above, we also evaluated a more advanced hardware arbitration scheme to reduce SRF bank conflicts.

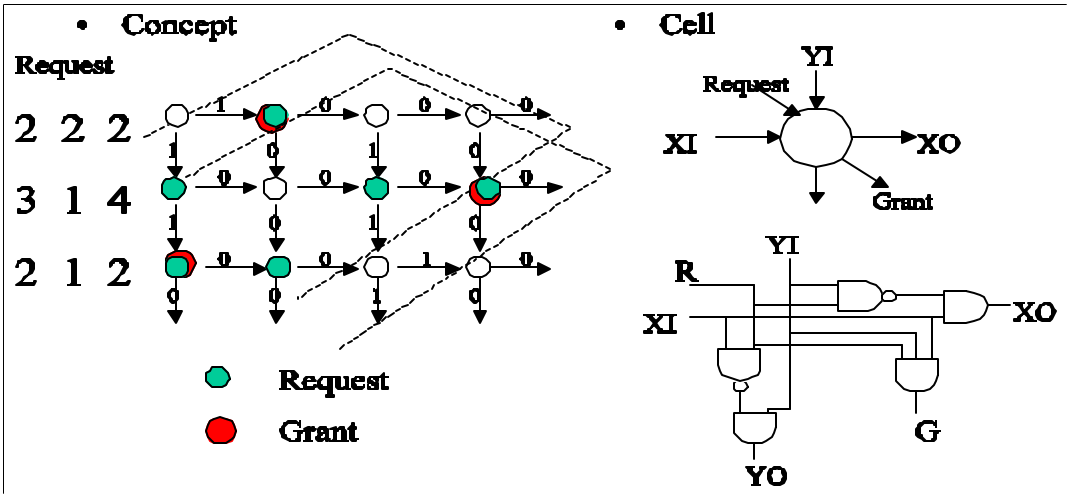
Basic assumption for this bank conflict reducing hardware is that there are only 'read only stream' or 'write only stream', and no 'read and write stream'. This assumption holds for the Imagine processor which is the base model for stream processor simulation. Due to this assumption, the order of processing can be neglected since a write stream will not be used as an input. Therefore, we can use buffers in each bank and issue SRF accesses out of order to avoid bank conflicts.

The previous figure shows an example of how this hardware works. For each cluster there's a buffer that contains the requests of the cluster. In the figure, the number in the request buffer means the number of the bank that is requested. In the first row, clusters request the data in bank3, 2, 2 and 1. So, there is a conflict in bank 2. To avoid this, the hardware arbiter will look up the next buffer and find the bank requests that have no



conflict. In this example, cluster 1 and 2 will get first element in the buffer, cluster 3 get second and cluster 4 get the third. A scoreboarding register will keep track of the data processing order. The data will be moved to the stream buffer in this location and the stream buffer will feed the cluster the same way that current Imagine works.

5.1 Allocator



To find non-conflicting requests, a wave front allocator is used. Each cluster requests one bank and the allocation grants only one requests per bank. In the figure below, the rows are the clusters and the columns are the banks. In each row, there should be only one grant such that the cluster is fed one data. There should also only be one grant in each column to avoid bank conflict. The selection for requests can be searched diagonally to satisfy these restriction. Figure below shows the implementation of this hardware. When the input of the cell is 0 (which means there'a already grant in that row or column) that request cannot be granted. For the fairness of the priority in each diagonal, a token for the priority is used to wrap around each cycle.

In the VLSI implementation of this allocator, only combinational logic is used. So the whole arbitration can be done in one cycle. If the number of banks is too big for this arbitration to be done within the cycle time, we can use lookahead logic, like the carry lookahead logic in adders.

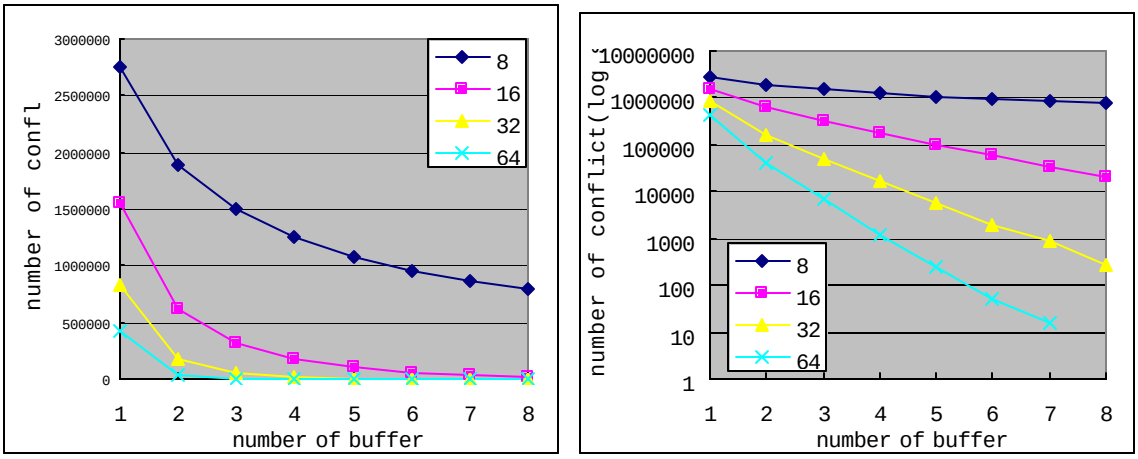
5.2 Description of Simulation

We simulated the performance of reducing bank conflicts as a parameter of number of banks and number of buffers at given cost. The wave front allocator and SRF indexing hardware were modeled in C. It counts the number of stalls caused by the bank conflicts while indexing the SRF even though it's not cycle accurate in modeling. The number of

banks, number of buffers, number of clusters(this one is not changed since we just assume current Imagine as the target) are parameterized to see the effect of these on the performance.

The area of the hardware is chosen to be the metric for the cost. From the layout of the Imagine and other circuits, area is estimated. The major component of the area was the bus between the buffers, since the length of this bus is about the sum of 8 ALU cluster width. This cost will be about the half of one set of inter-cluster communication buses. There is also the cost for the allocator which is proportional to the number of the banks, the SRF address switch, the Inter-Bank switch, and the switch control unit. There are also some more for request buffer, register for book-keeping and vertical connection from banks to the stream buffer, but this is very small compare to the other major cost unit. Based on the estimation of this cost, performance/cost is measured and a reasonable number of banks and buffers are found.

5.3 Performance/Cost Simulation

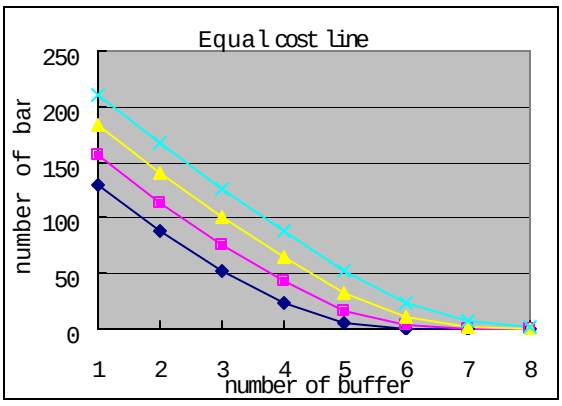


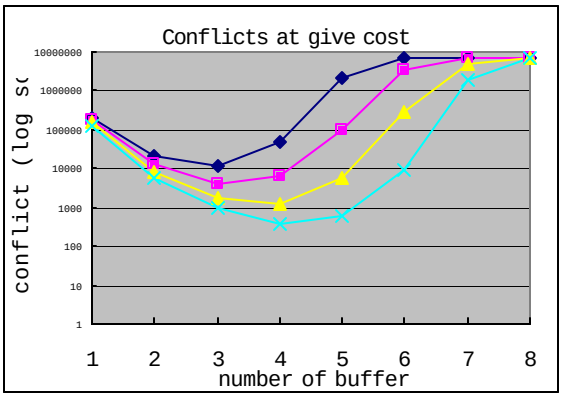
The figure above shows the result of reducing bank conflicts. An important thing to notice is that when there're enough number of banks (more than 16 or 32), the number of conflicts reduces quite rapidly as the buffer number increases. If there's no buffer, exponential increase of banks are needed to have the same result. This is reasonable result, since when we increase the number of buffer the possible combination of bank request will increase to (Number of Buffer) $^{(\text{Number of Bank})}$ .Even though the wave front cannot find the optimum solution for all the cases, it is quite close to the ideal case.

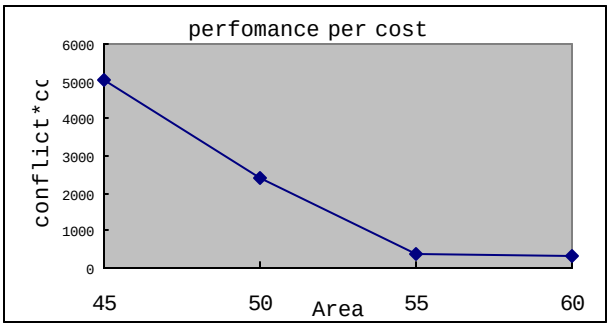
Given that there's lot of improvement in the bank conflict, the cost for this should be estimated. The cost analysis is below.

Bus from buffer to control unit	15bit * #buffer * bus_pitch * total length = 7280000 * (#buffer)
Allocator	(#bank * #cluster) * Cell size = 166400 * (#bank)
SCR	log(#bank) * #cluster * Cell size = 400000 * log(#bank)
Bus from control unit to buffer	log(#bank) * log(#buffer) * bus_pitch * total length = 1040000 * log(#buffer)
Switch control unit	log(#bank) * log(#cluster) * bus_pitch * Width/Cluster * #cluster = 2496000 * log(#bank)
SRF address switch	15bit * (#cluster) * total length * bit_pitch = 99840000
Inter-Bank switch	32bit *\#cluster * length * bit_pitch = 212992000
Etc : Request Buffer, Requester Buffer to SRF	
Total cost	= 7280000 * (#buffer) + 166400 * (#bank) * 1040000 * log(#buffer) + 3368000 * log(#bank) + 31283200 (Lamda square)

From the cost analysis it can be noticed that the cost function is more sensitive to the number of the buffer rather than the number of the banks. This is because it is required to have a long horizontal bus between buffers to arbitrate the requests for each buffer and to record the order of data processed.







The first graph of the figure above shows the lines that have equal cost. From the simulation the numbers of conflicts are plotted on those points. It shows some optimum points at given cost. This is the result of the two facts that were described previously. The effect of reducing bank conflicts and increasing of the cost complement each other and makes an optimum performance point in the equal cost line.

The last graph shows the performance/cost. The (area * number of conflict) is a metric that indicate (Cost/Performance). From the graph, it is shown that as we increase the cost, the performance per cost improves. But there is a saturation point. This point is about an area of 1/2 of one ALU cluster. So, it is quite reasonable not to increase the number of the banks or buffers that exceed this cost.

6 Conclusion

Although this project targets a very specific problem (unstructured mesh applications) for SIMD stream processors, it uncovers the advantage of data regularity and data reuse in enhancing streaming performance. Although unstructured mesh applications have variable record size, by separating core data information from neighbor information, we can kernelize the application. Because each node is mostly guaranteed to be visited several times, we can exploit data reuse by enhancing the effective bandwidth of the memory system through a cache or SRF indexing. Our simulation results show that either technique can be used to to reduce execution time by up to 75% for memory-limited applications that we studied with relatively little hardware overhead.

From the simulation point of view, we achieved most of the required simulations, except the imagine tool modification which would take at least a month to complete. Even

without the tool modification, our simulation results show consistent performance improvement with cache inclusion and SRF indexing. Because the entire project is based on static unstructured mesh applications where mesh data is read-only, it would be interesting to analyze the corresponding performance improvement of based on dynamic mesh structures. One additional advantage of SRF indexing that we have not explored in this project is the ability to use free stencils. Because of the free indexing nature, SRF indexing easily supports stencils provided that the maximum stencil distance does not exceed the stream size allocated in the SRF. Only a small modification in hardware must be made so that indexing becomes relative to the current stream element instead of absolute indexing.

Therefore, future research directions for this project would be to: complete the modification for Imagine tools, map dynamic mesh applications onto the Imagine architecture and use SRF indexing to support stencils.

Reference

[1]. Yuval Tamir, Hsin-Chou Chi, Symmetric Crossbar Arbiters for VLSI Communication Switches