Transitive Closure on the Imagine Stream Processor

Gorden Griem
Lawrence Berkeley National Laboratory
One Cyclotron Road
Berkeley CA 94720, USA
gorden.griem@tu-harburg.de

Leonid Oliker
Lawrence Berkeley National Laboratory
One Cyclotron Road
Berkeley CA 94720, USA
loliker@lbl.gov

ABSTRACT

The increasing gap between processor and memory speeds is a well-known problem in modern computer architecture. The Imagine system is designed to address the processor-memory gap through streaming technology. Stream processors are best-suited for computationally intensive applications characterized by high data parallelism and producer-consumer locality with minimal data dependencies. This work examines an efficient streaming implementation of the computationally intensive Transitive Closure (TC) algorithm on the Imagine platform. We develop a tiled TC algorithm specifically for the Imagine environment, which efficiently reuses streams to minimize expensive off-chip data transfers. The implementation requires complex stream programming since the memory hierarchy and cluster organization of the underlying architecture are exposed to the Imagine programmer. Results demonstrate that limited performance of TC is achieved primarily due to the complicated data-dependencies of the blocked algorithm. This work is an ongoing effort to identify classes of scientific problems well-suited for streaming processors.

1. ALL-PAIRS SHORTEST PATH

The problem of finding all the shortest paths in a graph is one of the most important optimizations in operations research as it arises in many applications, most notably network routing and distributed computing. Let $G = (V, E)$ be a directed graph with N nodes and M arcs, where the length of the arc(i,j) is denoted by l_{ij}. The Transitive Closure (TC), or *all-pairs shortest path*, computes the length of a minimum-length path between all pairs of nodes. The classical sequential approach for solving this problem is the $O(N^3)$ dynamic programming methodology of the Floyd-Warshall algorithm [5] shown Figure 1. The algorithm consisting of three nested loops where the inner two can be parallelized and executed in any order. A matrix *length* of dimension $N \times N$ holds the best known shortest distances between every pair of nodes. Initially, $length(i, j)$ contains the length of arc (i, j) if it exists, 0 if $i = j$, and ∞ otherwise. As the algorithm proceeds $length(i, j)$ is updated by considering all possible paths from i to j that pass through a growing set of nodes. This work presents a hardware-aware parallel implementation of the all-pairs shortest path problem on the Imagine processor.

```
for k = 1 to N do
  for i = 1 to N do
    for j = 1 to N do
      if (length(i,k) + length(k,j) < length(i,j) then
        length(i,j) = length(i,k) + length(k,j)
```

Figure 1: Floyd-Warshall (serial) algorithm

2. IMAGINE ARCHITECTURE

Imagine [6] is a programmable streaming microprocessor currently being developed at Stanford University. Stream processors are designed for computationally intensive applications characterized by high data parallelism and producer-consumer locality with little global data reuse. The general layout diagram of Imagine is presented in Figure 2. Imagine contains 48 arithmetic units, and a unique three level memory hierarchy designed to keep the functional units saturated during stream processing. The memory hierarchy consists of off-chip SDRAM (2.1GB/s), a 128KB stream register file (SRF) (25.6GB/s), and direct forwarding of results among arithmetic units via 528 local register files (LRF) (435GB/s) of which 256 per cluster are indirectly addressable. The architecture is centered around the SRF, which reads data from off-chip DRAM through a memory system interface and sequentially feeds the 8 arithmetic clusters. The local storage of the SRF can effectively reuse intermediate results (producer-consumer locality), allowing for the amortization of off-chip memory accesses. In addition, the SRF can be used to overlap computations with memory traffic, by simultaneously reading from main-memory while writing to the arithmetic clusters (double-buffering).

Each of Imagine's 8 arithmetic clusters consists of 6 functional units containing 3 adders, 2 multipliers, and a divide/square root. Imagine is a native 32-bit architecture; with support for performing operations on 16- and 8-bit data resulting in two and four times the peak performance respectively. A single microcontroller broadcasts VLIW instructions in SIMD fashion to all of the arithmetic clusters. This is in contrast to traditional vector architecture which issue a single instruction per cycle, counting on parallelism within each vector instruction to achieve high performance. The Imagine architecture emphasizes raw processing power

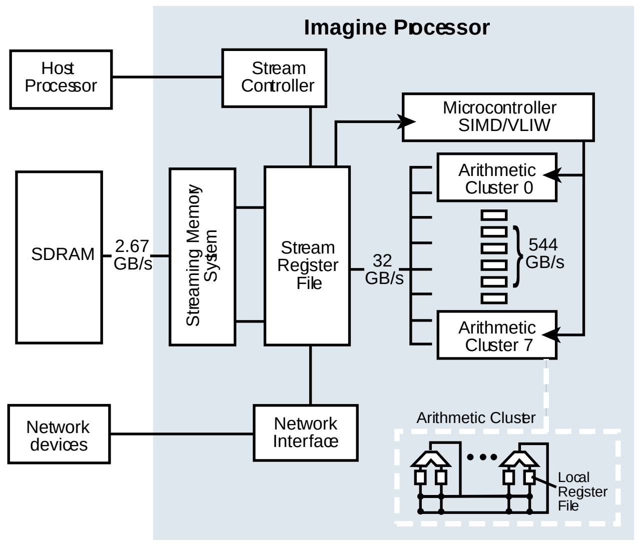

Figure 2: Overview of Imagine architecture

with a peak performance of 19.2 GOPS for 32-bit data; however, to achieve high performance expensive off-chip data transfers must be minimized.

Imagine supports the relatively new stream programming paradigm, designed to express the high degree of fine-grained parallelism necessary to effectively utilize the large number of functional units. The stream programming model organizes data as streams and expresses all computations as kernels. A stream is an ordered set of records of arbitrary (but homogeneous) data-objects. Kernels perform computation on entire streams, by applying potentially complex functions to each stream record in order. However, kernels cannot make arbitrary memory references and are limited to only accessing data from the SRF in a sequential fashion. The kernel memory reference restrictions allow the memory subsystem to effectively provide data to the large number of functional units. However, these memory access limitations increase programming complexity, especially for irregularly structured applications. This approach can be viewed as a generalization of vector computing with user defined, coarse-grained kernel operations replacing traditional vector instructions. In addition, chaining is also generalized through the use of the Stream Register File and producer-consumer locality.

In Imagine, two languages are used to express a program: StreamC is used to coordinate the streaming of data while KernelC is used to define the computational kernels to be performed on each stream record. Although syntactically like C, KernelC only supports a subset of C operations defined by Imagine's instructions. Most notably, no conditional branches exist. Separate stream and kernel compilers then map these two languages to the ISA of the stream controller and micro-controller respectively. The Imagine software environment allows for automatic code optimizations such as loop unrolling and software pipelining, as well as visual tools for isolating performance bottlenecks.

3. POWER3 ARCHITECTURE

For comparison purposes we present performance measurements on the IBM RS6000 POWER3. The POWER3 [3] is a 64-bit PowerPC implementation with a 32-byte backside L2 cache interface (private L2 cache bus), 32 visible and 24 renaming floating-point register, 64 KB L1 data cache accessible in one cycle, and 8192 KB L2 cache accessible in nine cycles. It has a peak execution rate of eight instructions per cycle and a sustained performance of four instructions per cycle.

Capable of executing up to four floating-point operations (two multiply-add instructions) per cycle on its two identical floating-point units (FPUs), the POWER3 puts emphasis on floating-point performance and memory bandwidth. The SMP-capable system design allows for concurrent operation of fixed-point instructions, load/store instructions, branch instructions, and floating-point instructions. Additionally the POWER3 supports out-of-order execution of instructions and speculatively executed branches.

On the IBM RS/6000 SP used for our analysis [2], the processor is clocked at 375 MHz, allowing a peak performance of 1500 MFlop/s if multiply-add instructions (FMA) are used (750 MFlop/s otherwise.) The POWER3 processor has on-chip hardware counters that allow the accurate gathering of performance data including the number of execution cycles and issued instructions. This data can be accessed by using the *Hardware Performance Measurement Toolkit* [1] without the introduction of measurement errors.

4. IMAGINE IMPLEMENTATION

The Imagine stream architecture is designed for algorithms that exhibit little data reuse, high data parallelism, and computational intensity. Examples which effectively utilize this architecture include graphics rendering, MPEG decoding, and BLAS3 computations. Unfortunately TC only satisfies two of these characteristics: the tiled algorithm is highly data parallel and compute intensive, but requires high global data reuse. Thus we do not expect TC to achieve near-peak performance on this architecture. However, critical insight is gained into the microarchitectural balance of

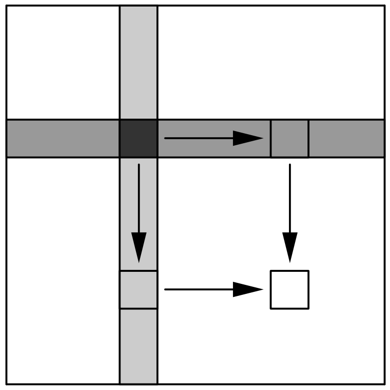

Figure 3: Illustriation of the tiled matrix in the 4^{th} iteration. The arrows indicate the data dependency. The black block is shown in black, blue blocks in dark grey, red blocks in light grey, and white blocks in white

the Imagine processor through a detailed analysis of the runtime components. Some of the algorithmic considerations required to effective utilize the Imagine architecture included: converting each computational block to a data stream, effectively reusing the limited number of addressable registers (256 per arithmetic cluster), managing stream register file reuse in the context of complex data dependencies, and minimizing expensive off-chip memory access operations.

This work develops a tiled implementation of the Floyd-Warshall algorithm [8] which effectively uses Imagine's three-tiered storage hierarchy; however blocking this problem in a streaming environment is nontrivial due to complex data dependencies. The tiled approach decomposes the domain into $B \times B$ tiles of size $\lceil \frac{N}{B} \rceil \times \lceil \frac{N}{B} \rceil$ and performs B block iterations. During the k^{th} block iteration, the $(k,k)^{th}$ tile (called *black block*) is first updated, then the remainder of the k^{th} row (*blue blocks*) and k^{th} column (*red blocks*). Finally the rest of the matrix (*white blocks*) is updated. It can be shown that this algorithm is provably correct and that the total processor-memory traffic is reduced to N^3/B. Given Imagine's 528 available local registers, the optimal block size of $\lceil \frac{N}{B} \rceil = 16$ is chosen for our experiments. With this block size the matrix is decomposed into one black block, $\lceil \frac{N-16}{16} \rceil$ red blocks, $\lceil \frac{N-16}{16} \rceil$ blue blocks, and $\left(\lceil \frac{N-16}{16} \rceil\right)^2$ white blocks.

Due to differing data dependencies, each of the four colored block types uses a special computational kernel to perform the updates. Most notably, a black block depends only on itself, a red/blue block on itself and the updated black block, and a white block on the updated red block in the same row and the updated blue block in the same column (see Figure 3. Although the same number of arithmetic operations are executed for each color class, the runtimes vary significantly due to these data dependencies. For a blocksize of 16×16, only one block fits into the local register files. However, up to three blocks worth of data may be required to update certain colors; thus it was necessary to modify the loop-order of the classical Floyd-Warshall algorithm.

The limited 2.1 GB/s bandwidth between Imagine and the host processor can be the most significant source of performance bottlenecks. Therefore an intelligent implementation should minimize the volume of data transfer between Imagine and the off-chip DRAM, by optimizing stream reuse and thus leveraging producer-consumer locality. For TC, the minimum memory transfer between host processor and Imagine can be calculated in the following way: Each block has 256 single-precision floating-point entries, resulting in total of 1024 bytes per block or 8192 bytes per stream (a stream contains eight blocks.) The number of iterations is $\lceil \frac{N}{16} \rceil$ and at each step the host processor requires a fully updated copy of the matrix. Taking the colored blocks into account, we compute a lower memory bound (in bytes) as:

$$\text{mem} = 8192 \cdot \left\lceil \frac{N}{16} \right\rceil \cdot \left(1 + 2 \cdot \left\lceil \frac{N-16}{8 \cdot 16} \right\rceil + \left\lceil \frac{(N-16)^2}{8 \cdot 16} \right\rceil \right)$$

Implementing effective stream reuse can be rather challenging, since stream elements are distributed to the arithmetic clusters in a round-robin fashion, and stream element reordering within the SRF is limited to strided and offset-based addressing. Therefore the kernel implementing must be carefully written to facilitate stream reuse. As a result, the kernel may consume stream elements using complicated offset calculations to preserve proper element ordering. Other techniques such as strategically duplicating stream elements are also used. Data transfer to the host processor for stream reordering on the off-chip DRAM is performed only as a last resort, due to its high overhead.

```
for k=0 to 255 do
  in_black >> black(k);
for k=0 to 15
  for i=0 to 15
    for j=0 to 15
      if (black(i,k)+black(k,j) < black(i,j)) then
        black(i,j) = black(i,k) + black(k,j);
for k=0 to 255 do
  out << black(k);
```

Figure 4: Pseudocode for black block update

The Floyd-Warshall algorithm extensively uses addressable registers in the three nested loops for index-based data references. In order to directly name these registers, the loops must be unrolled. Although this is practical for the inner most loops, it is infeasible to unroll all three loops as the kernels would become unreasonably long ($16^3 = 4096$ comparisons). Thus only a subset of the named registers can be used in the inner-most loop, as the implementation is bound by the 256 indirectly addressable registers. Judicious management of these limited registers is therefore required to optimize performance.

4.1 Black block Update

To update the black block efficiently, all 256 block elements are read and stored into LRF, as the entries are dependent on each other. The Floyd-Warshall algorithm is then applied to the data, with results saved in the form of an output stream. Figure 4 shows the pseudocode for the black kernel. Since there exists just one black block, a single kernel call (and ALU-cluster) is required to perform the update calculation. However, our implementation sends all eight ALUs an individual copy of the black block, allowing the same update to be performed (redundantly) on each cluster: as a result, the output stream consists of eight identical copies of the updated black block. These output streams are then reused during red and blue block updating, thus minimizing overhead by eliminating expensive host processor traffic. This demonstrates one example of leveraging SIMD parallelism to optimize stream reuse.

4.2 Red and Blue Block Updates

For the red and blue updates, eight blocks are updated in parallel on Imagine's ALU clusters. However, fewer block may be processed in the final step, as the red/blue blocks may not be evenly divisible by eight. For those cases certain blocks are distributed to more than one cluster, allowing for better reuse of the generated output stream.

The red/blue calculations are data dependent on themselves, as well as the updated black block. As described in Section 4.1 the special structure of the black calculation allows

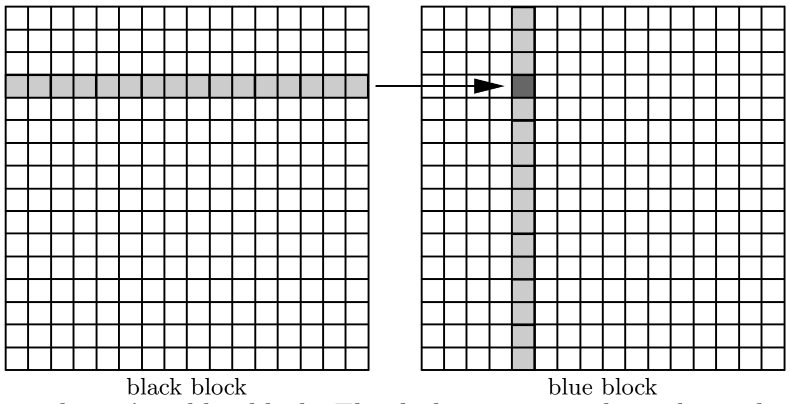

Figure 5: Data dependence in a blue block: The dark grey entry depends on the light grey entries

the blue and red kernels to receive the updated black data stream directly. However since only one blocksize of 16×16 fits into the indirectly addressable registers, data dependency analysis is used to optimally determine which block is to be kept in local registers. Figures 5 and 6 show the data dependencies for entries in the blue and red blocks respectively.

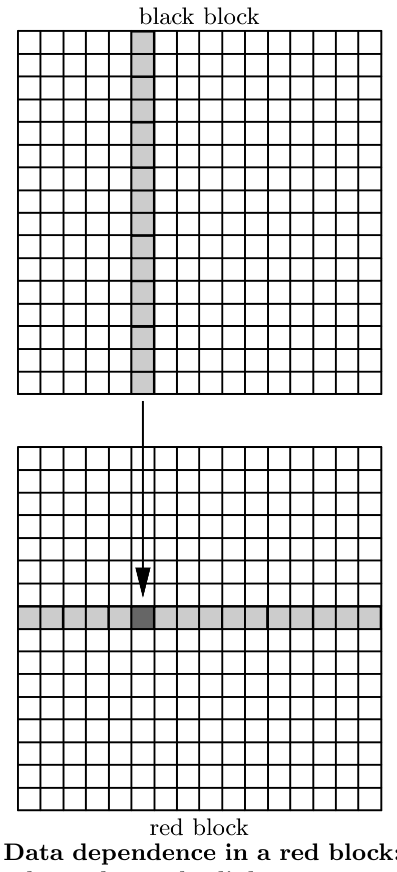

Figure 6: Data dependence in a red block: the dark grey entry depends on the light grey entries

For the blue blocks, it is possible to preserve the original (k,i,j) loop ordering. Since a single entry of the black block is only needed in the innermost loop, only one register is required. This register is assigned a new value after the completion of each j-loop. The blue block, is completely stored in local registers, thus a total of 256 addressable and

```
for k=0 to 255 do
  in_blue >> blue(k);
for k=0 to 15 do
  for i=0 to 15 do
    in_black >> black(i,k);
    for j=0 to 15 do
      if ((black(i,k) + blue(k,j)) < blue(i,j)) then
        blue(i,j) = black(i,k) + blue(k,j);
for k=0 to 255 do
  out << blue(k);
```

Figure 7: Pseudocode blue block update

```
for k=0 to 255 do
  in_red >> red(k);
for k=0 to 15 do
  for j=0 to 15 do
    in_black >> black(k,j);
    for i=0 to 15 do
      if (red(i,k) + black(k,j) < red(i,j))
        red(i,j) = red(i,k) + black(k,j);
for k=0 to 255 do
  out << blue(k);
```

Figure 8: Pseudocode for red block update

1 named registers are required for this kernel. Figure 7 shows the pseudocode for updating the blue blocks.

The data dependence pattern for the red block is shown in Figure 6. Due to dependence constraints the red kernel was implemented by interchanging the (i,j)-loop ordering. This can performed while preserving algorithmic correctness. The entry $black(k, j)$ is only needed in the inner loop after the loop interchange, as seen in the pseudocode of Figure 8. Thus, as in the blue block, 256 addressable and 1 named registers are required for the kernel computation.

4.3 White Block Update

Each white block entry calculation is dependent on both the updated red block in the same column and the updated blue block in the same row. For this case, the limited number of registers is not sufficient for caching all of the necessary blocks, since only a single indirectly addressable block fits into the LRF. Therefore it was necessary to develop a specialized algorithm to address the limited register space while minimizing off-chip data transfers.

```
for k=0 to 255 do
  in_blue >> blue(k);
for i=0 to 15
  for k=0 to 15
    in_red >> red(i,k);
  for j=0 to 15
    in_white >> white(i,j);
    for k=0 to 15
      if ((red(i,k) + blue(k,j) < white(i,j))
        white(i,j) = red(i,k) + blue(k,j);
    out << white(i,j);
```

Figure 9: Pseudocode for white block update

First observe that for the white kernel the order of all three loops in the Floyd-Warshall algorithm can be exchanged while preserving correctness. This is because no block set is data dependent on the output of the white update. The pseudocode in Figure 9 shows the loops as the are executed in an (i,j,k)-order. Due to the modified loop ordering, the $(i,j)^{th}$ white entry only needs to be read once for each k-loop. Furthermore, the i^{th} row of the red block can be read into named registers for one (j,k) iteration and then safely overwritten with other data. The blue block, however, has to be kept in memory for the duration of the kernel call. As a result, the total number of registers needed to update a white block is 256 addressable and 17 named registers per cluster, easily fitting into the LRF of the Imagine SIMD clusters.

4.4 Superblocks

Due to the inherent SIMD parallelism used to operate Imagine's eight ALU clusters, the concept *superblocks* is introduced. A superblock consists of $t \times t$, $8 \times t$, or $t \times 8$ white blocks, $t \in \{1, \ldots, 8\}$, with t preferably equal to eight. For each possible superblock size, a different updating scheme for optimal stream reuse is implemented. As each ALU cluster performs the update on one block, eight blocks are updated in parallel whenever data dependencies allow. For conciseness we omit a description of all updating schemes; however, three important examples are show below.

4.4.1 8×8 superblock

The most common case for large matrices is the superblock of size 8×8. The output stream containing the blue blocks associated with the superblock is directly reused, but a special variant of the red stream has to be created. In this modified stream, every entry is contained twice such at an element at position $(k \cdot 16 + n)$ is also at position $(k \cdot 16 + 8 + n), k \in \mathbf{N}, n \in \{0, \ldots, 7\}$; thus each cluster receives two copies of every element. A special kernel call is used to create the modified red stream, such that no off-chip data transfer is necessary. This scheme allows the reuse of a red stream, with a different offset for each kernel call. A total of eight kernel calls are needed to find the shortest paths for all 64 white blocks of the superblock.

During the first kernel call, with a given offset of 0 for the red stream, ALU0 calculate the shortest paths for the 0^{th} white block, ALU1 calculates the 9^{th} white block and so on. During the second kernel call, with the red stream offset by 1, ALU0 computes the 8^{th} white block, ALU1 updates the 17^{th} white block, etc. This continues until all 64 blocks have

1	8	7	6	5	4	3	2
2	1	8	7	6	5	4	3
3	2	1	8	7	6	5	4
4	3	2	1	8	7	6	5
5	4	3	2	1	8	7	6
6	5	4	3	2	1	8	7
7	6	5	4	3	2	1	8
8	7	6	5	4	3	2	1

Figure 10: Kernel call iteration for updating white blocks in 8×8 superblock

been update through the eight kernel calls. Figure 10 shows the iteration number where each white block is updated at in a 8×8 superblock.

4.4.2 5×5 superblock

A 5×5 superblock contains 25 white blocks, and therefore requires a total of $\lceil 25/8 \rceil = 4$ kernel calls. Computation is performed on the first row and the first three entries of the second row at the first kernel call; on the third row and the first three entries of the fourth row at the second kernel call; on the fifth row at the third kernel call; and on all other white blocks at the fourth kernel call. Figure 11 presents an illustration of the kernel calls.

Unfortunately, the blue stream can only be reused for the first three kernel calls, while the red stream cannot be reused at all. To allow reusage, the blue stream gets the 0^{th} blue block assigned to positions 0 and 5; the 1^{st} blue block assigned to positions 1 and 6; the 2^{nd} blue block assigned to positions 2 and 7; and the 3^{rd} to 5^{th} blue block assigned to positions 3 to 5. The blue kernel is then called, resulting in an output stream with the same data layout, and thus allowing the reuse of the blue output stream for the first three iterations. For the fourth kernel call, a new stream has to be generated with the 3^{rd} blue block at positions 0 and 2, and the 4^{th} blue block at positions 1 and 3. This stream is then transferred to Imagine.

The red stream, on the other hand, has to be generated and transferred to Imagine for every kernel call. For the first kernel call, the 0^{th} red block is assigned to positions 0 to 4 in the stream, and the 1^{st} red block to positions 5 to 7; in the second kernel execution, the 2^{nd} red block is assigned to positions 0 to 4, and the 3^{rd} red block to positions 5 to 7; on the third kernel call, the 4^{th} red block is assigned to positions 0 to 4; finally, for the fourth, the 1^{st} red block is assigned to positions 0 to 1 in the stream and the 3^{rd} red block is assigned to positions 2 to 3.

4.4.3 $t \times 8$, $8 \times t$ superblock

For a $t \times 8$ superblock, the most efficient method is to update one row at a time, since there are 8 blue blocks and t red blocks associated with the superblock. This scheme requires t kernel calls and reuses the blue stream; however, it is necessary to create a red stream for each i^{th} kernel call that contains a copy of the i^{th} red block in every entry of the stream.

1	1	1	1	1
1	1	1	4	4
2	2	2	2	2
2	2	2	4	4
3	3	3	3	3

Figure 11: Kernel call iteration for updating white blocks in 5×5 superblock

For a $8 \times t$ superblock, there are t blue blocks and 8 red blocks associated with the superblock. The most efficient computation approach is to update one column at a time. Therefore a total of t kernel calls are needed to perform the calculation for all white blocks in the superblock. The red stream can be reused; however the blue stream has to be created for each iteration, such that the i^{th} kernel call consists of eight copies of the i^{th} blue block.

These superblock implementations demonstrates the complexity of code development in a streaming environment where the memory hierarchy and cluster organization is exposed to the programmer. Improvement in the quality of the software development tools and abstracting lower level details of the hardware - work currently in progress[4] - will be essential in bringing the stream programming model to the wider community.

5. POWER3 IMPLEMENTATION

For the reference scalar version of Transitive Closure on the IBM POWER3, we implement the cache-efficient tilted algorithm described in [8]. Since POWER3 performance is presented as a reference point, scalar code optimizations are not discussed in great detail.

The POWER3 algorithm is similar to the Imagine version, where the matrix is divided into black, blue, red and white blocks. After an extensive search, we found that a blocksize of $N = 4$ achieved the best performance. To allow a fair comparison with Imagine, single-precision floating-point arithmetic was used.

The POWER3 TC code is implemented in C and uses hand-unrolled loops. Intermediate results are saved in local registers to allow optimal instruction scheduling/reordering. The IBM *xlc* compiler is used with the optimization flags *-O5* and *qarch=pwr3*. This allows the generation of 32-bit code using architecture-specific instructions and instruction scheduling optimized for pipeline length, as well as minimized pipeline stalls.

6. PERFORMANCE EVALUATION

We extensively gathered Imagine performance results using the cycle-accurate simulator *ISim* for all problem set sizes of N in the range of $N = 32$ to $N = 512$ (in multiples of 16). The total runtime trend shows the general behavior of a O(N^3) algorithm; however, performance irregularities can be seen due to varying stream reusage efficiencies of different superblock sizes. For large N, performance numbers stabilize; thus for $N > 512$, we expect performance to behave similarly to $N = 512$.

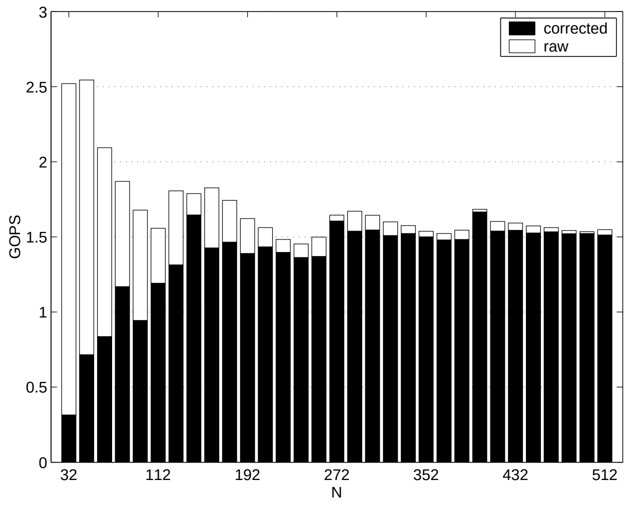

Figure 12: Billions of raw and corrected operations per second on Imagine

To evaluate performance we measure the number (in billions) of operations per second (GOPS). Since all eight arithmetic cluster execute each given kernel, there are cases when some of ALUs perform redundant work to improve stream reuse. Although this artificially inflates the number of operations performed, it does reduce the overall runtime. We therefore denote *raw* GOPS as the total number of operations (including redundant ops), and *corrected* GOPS as only those operations generating new information. For comparisons with other architectures, the corrected GOPS is the proper metric of comparison.

The number of raw and corrected GOPS is presented in Figure 12. Notice that the maximum raw operations is about 2.5 GOPS for a matrix size of 48×48; however, the maximal corrected value is only 1.66 GOPS. As can be seen in Figure 12, raw and corrected GOPS differ greatly for small matrices, while for large matrices the two values converge. This is because decomposing large matrices into colored blocks, results in relatively few “idle” clusters. Given Imagine’s peak performance of 19.2 GOPS, the Transitive Closure algorithm achieves only 8.7% of peak. This is more than a factor of 10 slower than media applications successfully implemented on the Imagine system ([6], [7]).

As seen in Figure 13, our specialized Imagine implementation reuses data streams aggressively and results in memory transfers near the theoretical minimum. Nonetheless, it is the total volume of off-chip data movement that causes significant performance degradation. Figure 14 shows that only a small fraction of the total cycles (23%-37%) are actually accounted for by the kernel execution. The remaining cycles are mostly consumed for host and SRF data transfers, while a small percentage is necessary for loading the microcode.

Although seemingly counterintuitive, the especially low fraction of cycles spent on kernel execution for large matrices can explained as follows: The black, red, blue, and white kernels consume 45745, 41649, 41649, and 15456 cycles respectively (due to different data dependencies). Since the

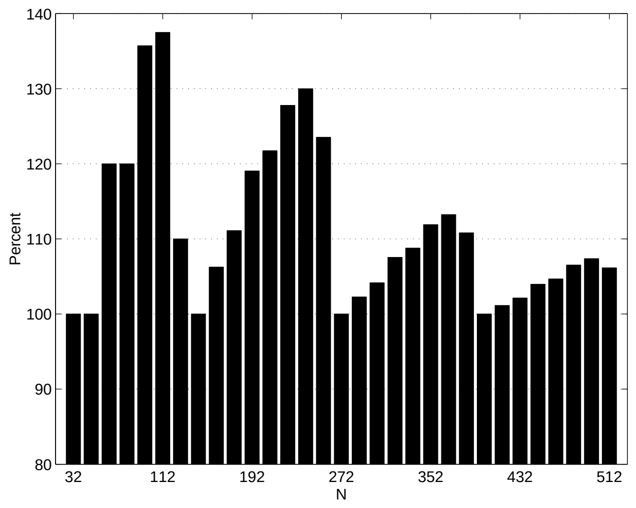

Figure 13: Percentage of actual versus theoretically minimal memory transfers

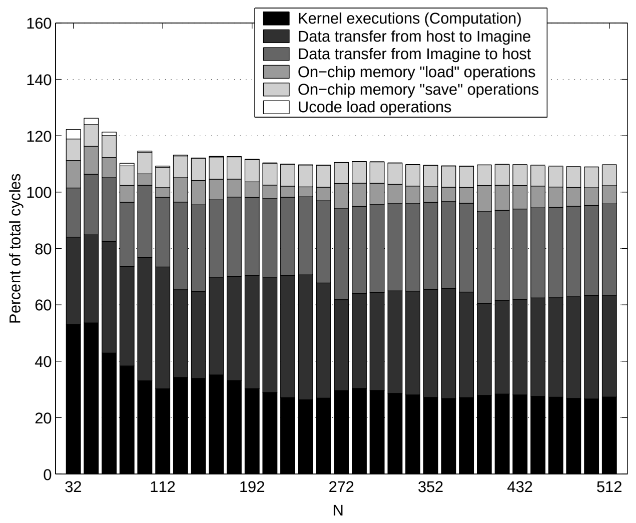

Figure 14: TC overheads on Imagine for varying N. Totals exceed 100% due to operation overlap

number of white blocks increases with N^2 compared with N for red/blue blocks, the average number of cycles spent on updating a block per iteration depicted in Figure 15 decreases with increasing N. The total number of bytes transferred between host processor and Imagine per block also decreases with increasing N as stream reuse becomes more efficient (Figure 16); however it decreases slower than the number of cycles needed to update a block. Therefore the ratio between bytes transferred to host and consumed cycles increases with N, as seen in Figure 17. This results in more cycles spent on memory transfers even as stream reuse becomes more efficient.

Overall, due to the data transfers and complex data dependencies, our optimized hardware-aware implementation achieves less than 10% of the sustained peak performance on this platform, confirming that the Imagine architecture is not well-suited for algorithms requiring global data reuse.

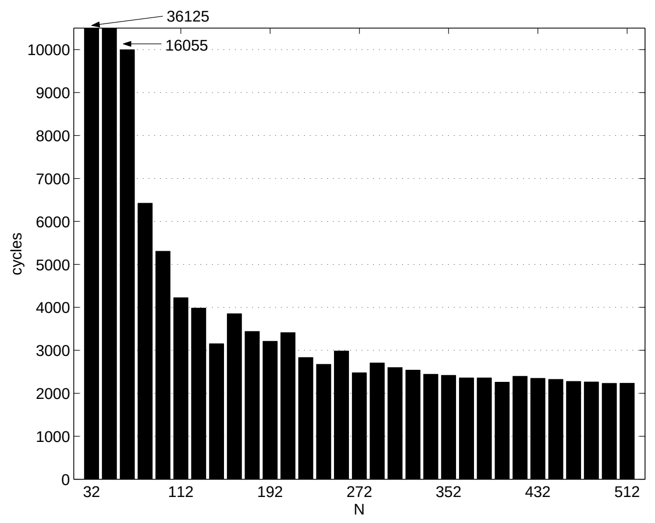

Figure 15: Average computational cycles

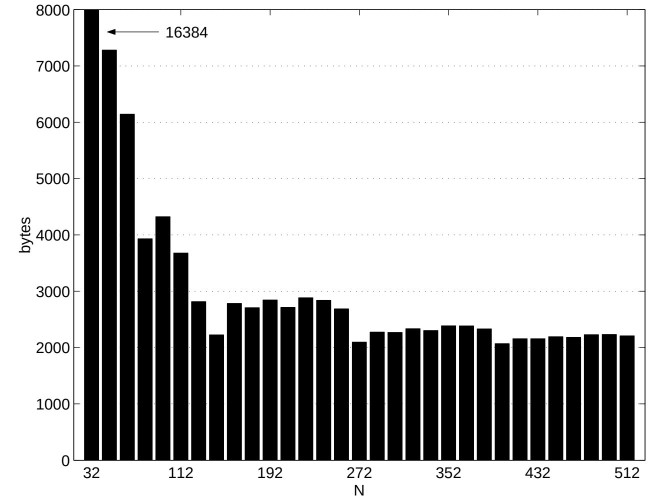

Figure 16: Average number of bytes transferred between Imagine and host processor

On the POWER3, all hardware counter measurement were gathered using the highly accurate *Hardware Performance Monitor Toolkit* [1]. The achieved floating point operations per second for the POWER3 architecture can be seen in Figure 18. Since only floating-point add operations are used, the maximum achievable rate of floating-point instructions is 750 MFlops/s. Observe that the cache-based version achieves more than 50% of theoretical peak. Thus, on average, more than one floating point instruction is executed per cycle, indicating that instruction-level parallelism is effectively exploited.

A cycle comparison of the Imagine implementation with the optimized serial implementation on the IBM POWER3 is shown in Figure 19. At best Imagine achieves only a 20.6% reduction in the required cycles, even though it exceeds the POWER3 by more than $13X$ in raw computational power. Additionally, the memory transfer overhead is particularly expensive for small N, resulting in poor performance relative to the POWER3. These results confirm that although TC is a computationally intensive algorithm, its tiled implemen-

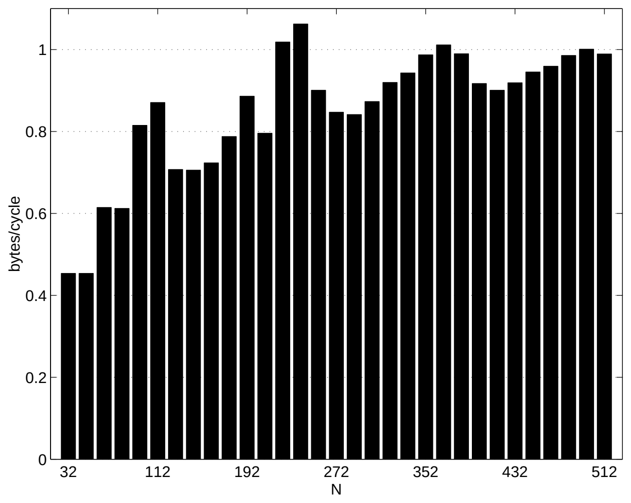

Figure 17: Bytes transferred between Imagine and host processor vs. computational cycles

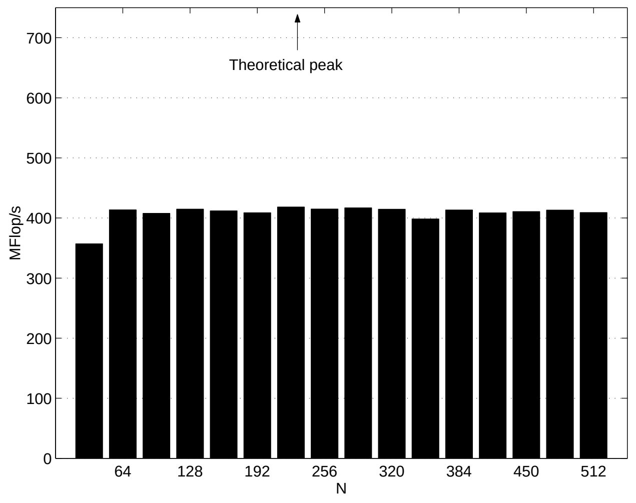

Figure 18: POWER3 performance (in MFlop/s)

tation is memory-bandwidth bound on Imagine. However, Imagine consumes about $\frac{1}{10}$ the energy of a POWER3 processor, making the implementation on Imagine up to twelve times more energy-efficient. Therefore Imagine is a good candidate for energy-limited applications like wireless network routing on mobile devices.

7. CONCLUSION

This paper examined an optimized implementation of Transitive Closure on the Imagine stream processor. We presented a tiled version of this important algorithm and described the challenges of code development in a streaming environment where the details of the underlying hardware are exposed to the programmer. Extensive runtime analysis was presented to understand the performance characteristics and architectural bottlenecks. Overall the Imagine processor has tremendous computational potential and achieves excellent performance for applications well-suited for its architecture; however, off-chip memory transfers must be minimized. Our work shows that if random data access requirements are sufficiently large, even computationally intensive algorithms with hardware-aware tiled implementations do not guarantee high performance.

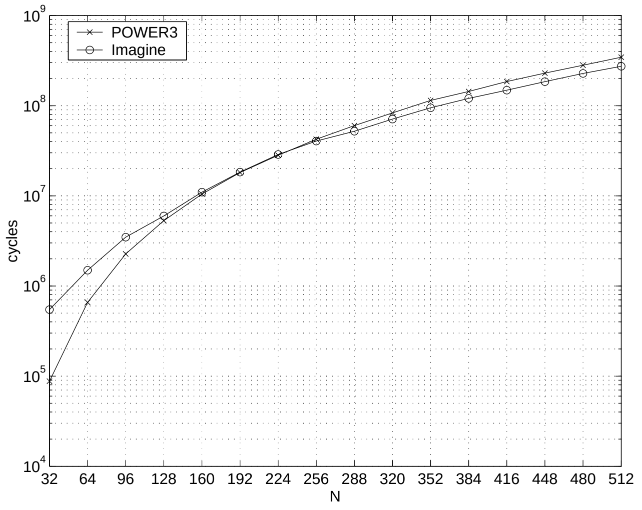

Figure 19: Imagine vs. POWER3 (in cycles)

Acknowledgements

The authors would like to gratefully thank Stanford's Imagine group for providing access to the Imagine simulator. We also thank Parry Husbands for his contributions. The authors were supported by Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC03-76SF00098.

8. REFERENCES

[1] *The HPM Toolkit.* http://www.alphaworks.ibm.com/tech/hpmtoolkit.

[2] *NERSC IBM SP.* http://www.nersc.gov/computers/SP.

[3] S. Andersson, R. Bell, J. Hague, H. Holthoff, P. Mayes, J. Nakano, D. Shieh, and J. Tuccillo. *RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide.* IBM Redbook. IBM, http://www.redbooks.ibm.com, 2003.

[4] W. Dally, P. Hanrahan, and R. Fedkiw. A streaming supercomputer. In *Whitepaper*, September, 2001.

[5] R.W. Floyd. Algorithm 97: Shortest Path. *Communications ACM*, 5:345, 1962.

[6] U.J. Kapasi, W.J. Dally, S. Rixner, J.D. Owens, and B. Khailany. The Imagine Stream Processor. In *Proc. IEEE International Conference on Computer Design*, pages 282–288, 2002.

[7] J. D. Owens, S. Rixner, U. J. Kapasi, P. Mattson, B. Towles, B. Serebrin, and W. J. Dally. Media processing applications on the imagine stream processor, 2002.

[8] M. Penner and V. K. Prasanna. Cache-friendly implementations of transitive closure. In *Proc. of International Conference on Parallel Architectures and Compiler Techniques*, 2001.