THE MATRIX MULTIPLIED AND STREAMIFIED
CS 343 Final Report, Spring 2003

Abhishek Das, François Labonte, and Suzanne Rivoire

I. INTRODUCTION

In this report, we examine different techniques for data layout of a matrix-matrix multiplication program on the Imagine streaming architecture [1]. Determining the correct way to arrange data for this problem is one step towards the larger goal of adapting generic codes to the stream programming model. This general goal requires work on several fronts. Initially, we intended to explore the question of modifying a C compiler to identify streams and kernels, but we quickly discovered that this process was straightforward and mechanical for the well-defined programs we wanted to explore, with most of the work lying in the learning curve for the compiler infrastructure. Consequently, we focused instead on the question of blocking computation for the Imagine architecture once streams and kernels have been identified. The Imagine team had already addressed the question of blocking convolution, but matrix-matrix multiplication remained an open problem.

Our goal was to determine whether blocking for Imagine’s software-managed stream register file (SRF) would be substantially different from blocking for a traditional reactive cache. Our results show that, as in cache-based architectures, blocking outperforms simpler methods when the matrix is too large to fit in intermediate storage. But at some point the number of blocks and partial results generated use so much bandwidth that they overwhelm the advantage of the reuse in blocking, making the simpler method better.

Section 2 of this report sums up approaches to matrix-matrix multiplication on conventional architectures, while Section 3 highlights key aspects of the Imagine architecture. Section 4 describes the Imagine software tool chain and software techniques for improving performance on Imagine. Section 5 describes our approach to matrix multiplication on Imagine, and Section 6 presents results of our experiments.

II. MATRIX MULTIPLY ON CONVENTIONAL ARCHITECTURES

A. Naïve Approach

Matrix-matrix multiplication is a highly parallel and computationally intensive application, with many operations per data item. However, a naïve approach to this application can lead to poor cache performance despite the high amount of data reuse.

Fig. 1 illustrates the intuitive approach to matrix-matrix multiplication. In this approach, we first load a row of the first matrix and a column of the second to produce the first element of the output matrix. Then we load the second column of the second matrix, reusing the first row of the first matrix. This approach yields good performance as long as the input row and column fit in the cache.

Suppose only the shaded portion of the first row of matrix A, as shown in Fig. 1, fits in the cache. In this case, this shaded portion will be loaded and multiplied by the first column of matrix B. Next, the remaining (unshaded) portion of matrix A’s first row will be loaded and multiplied by matrix B’s first column, and the first output result will be stored. To compute the next element, the gray portion will again be loaded. Consequently, no data elements will be reused from one computation to another, resulting in disastrous cache performance.

B. Blocked Approach

An alternate approach is to divide each matrix into blocks, as in Fig. 2, and compute partial sums for each block to be added in a final step. Because of the addition of partial sums, this algorithm is less efficient than the naïve approach for small matrices, but dramatically increases locality for large matrices for which the other approach fails entirely [2].

III. THE IMAGINE STREAM ARCHITECTURE

The Imagine stream processor, depicted in Fig. 3, is designed for data-parallel applications that require high
memory bandwidth. This section describes the computational and memory resources of the Imagine processor.

The computational resources of Imagine are organized into clusters of ALUs, as shown in Fig. 3. Each cluster contains three adders, two multipliers, one divide/square root unit, a scratchpad memory, and an intercluster communication unit. Imagine’s eight clusters run identical statically scheduled VLIW instructions. Imagine is designed to exploit instruction-level parallelism within its clusters and data-level parallelism in the SIMD execution across clusters [1].

Fig. 3. Block diagram of the Imagine stream processor. Each cluster executes the same statically scheduled VLIW instruction in SIMD. The stream register file is the software-managed level of intermediate storage for which we are blocking.

Feeding Imagine’s computational resources requires high memory bandwidth, which Imagine achieves through its bandwidth hierarchy. Each cluster contains local register files which can provide 435 GB/s of bandwidth. The software-managed SRF can provide the clusters with 26 GB/s of bandwidth. Finally, the off-chip memory can supply the SRF with 2.1 GB/s of bandwidth.

IV. SOFTWARE TOOLS AND TECHNIQUES

This section explains the Imagine programming model, the software tools used to compile Imagine programs, and the software techniques used in blocking.

A. Programming Model

The Imagine programming model organizes computation into kernels connected by streams, which are sequences of records. Kernels are small, computationally intensive programs that run repeatedly on Imagine’s clusters until their input streams are exhausted. Imagine kernels are coded in a dialect of C called KernelC. Kernels are permitted to access local variables and their input and output streams, but they are not allowed to make arbitrary memory references.

The application-level program is written in StreamC, another C dialect. StreamC code is run on the host processor and orchestrates the movement of data to and from Imagine. Kernels are initiated through StreamC kernel function calls [3].

B. Tools

StreamC and KernelC each have a different compiler; kernels are compiled with IScd, and StreamC is compiled with IStream.

IScd, the kernel scheduler, generates a microcode VLIW program for each kernel. It supports modulo software pipelining and loop unrolling.

IStream, the stream scheduler, determines the memory and SRF allocation for the streams and uses strip-mining or double-buffering (described in the next section) as necessary [4]. It executes kernels in parallel with loading subsequent streams to the SRF. Part of our work was to modify IStream to overlap two parallel memory transfers with kernel execution with adequate loop unrolling.

Stream scheduling is a profile-based compiler extension that manages the SRF better than a run-time technique, such as stream caching. Allocating buffers based on a profile of the program enables it to consider all stream accesses, not just past accesses; therefore, only data that is ejected from the SRF and will be reused later needs to be stored to memory. The stream scheduler arranges streams to make efficient use of the SRF and tries to assign derived stream accesses to adjacent buffers.

C. Software Techniques

This subsection describes two techniques used to manage the SRF: strip-mining and double-buffering.

Strip-mining, illustrated in Fig. 4, involves dividing a large initial input stream into smaller batches so that the intermediate streams produced for each batch will fit in the SRF. Strip-mining applies the series of stream operations to a small portion of the initial input to produce a small portion of the final output, such that the output of every stream operation fits in the SRF. It then applies the series to another small

Fig. 4. Strip-mining a large stream into smaller pieces. The figure on the left shows kernels (in yellow) operating on large streams (in blue). The figure on the right shows the stream broken down into smaller pieces by strip-mining, which will eventually be rejoined.
portion of the initial input to produce another small portion of
the final output, and so on until all of the initial input has been
processed. The size of this portion of the initial input is called
the strip size. Since most stream programs operate on inputs
that are larger than the SRF, this optimization is essential for
good performance.

Fig. 5 Double-buffering to cycle through a large input stream.

Double-buffering is another technique used to handle large
streams. If all of the streams accessed by a given operation
cannot fit in the SRF, stream scheduling assigns one or more
large streams to smaller buffers. At run time, double-buffering
is used to cycle those streams through their buffers. Double-
buffering cycles portions of a large stream through two halves
of a smaller buffer. Initially, the first portion of the stream is
loaded into half-buffer 1. Then, the clusters read the contents
of half-buffer 1, while a new portion of the stream is loaded
into half-buffer 2. When the clusters are
done with the portion of the stream in
half-buffer 1, the half-buffers swap roles.
A new portion of the stream is loaded into
half-buffer 1, while the processing
elements read the contents of half-buffer
2. The half-buffers repeatedly swap roles
until the entire stream has been read.

V. MATRIX MULTIPLY ON IMAGINE

A. Naïve Approach
To implement the naïve approach on
Imagine, we loaded a row of matrix $A$
and two columns of matrix $B$ so that we could
exploit instruction-level parallelism by
using both multipliers in each cluster.

We tried two implementations of the
matrix-matrix multiplication kernel
following this naïve approach. The
difference between the implementations
was the handling of the two output elements being produced
on each iteration of the kernel. In our first implementation,
we used a variable in the microcontroller register file.
However, an architectural prohibition against transferring
variables from this file to the SRF in parallel with kernel
computation was causing frequent stalls between kernel
executions. Our second, superior approach used conditional
streams [5] to reduce the 8 cluster outputs to 2. Fig. 6, on the
next page, shows a kernel schedule for this approach.

For matrix sizes that fit in the SRF, this approach performs
acceptably. For matrix sizes that exceed the size of the SRF,
the rows and columns are loaded in strips (strip-mining).
Double buffering allows us to load the next strip while
executing the kernel on the current strip. However for large
strip, the kernel execution is really short compare to strip
loading time and actually stalls 90% of the time waiting for
the next strip (Fig 10).

B. Blocked Approach
The size of the blocks was the most important consideration
for this approach. A block of the matrix $A$ will be reused as we
cycle through all the blocks of matrix $B$. Both the block from
the matrix $B$ and the output block of partial products will be
double-buffered so that the loading of the next block and
storing of previous results can be overlapped with
computation for the current blocks. We selected a block size
of less than a fifth (6553 words) of our intermediate level of
storage, the stream register file (32K words).

To exploit data-level parallelism, the 8 SIMD clusters work
on 8 rows and 8 columns at a time. Each cluster reads a row
element of matrix $A$ and a column element of matrix $B$. The
row elements are broadcast to other clusters. The inner loop
of the kernel is limited by the communication to other clusters,
as shown in Fig. 7 (next page).
C. Comparison

The application is clearly bandwidth-limited, since the kernels are completely overlapped by memory operations. When the operand matrices fit inside the intermediate level of storage, the simple method outperforms the blocked one, since the blocked method has the extra step of combining the partial results. Fig. 8 shows that for square matrices with side greater than 128, which no longer fit in the SRF, the blocked matrix method performs twice as fast.

As the matrix size increases, and the number of blocks required for the blocked method increases, this increases the number of partial result matrices. This rise in partial results, that further need to be combined in the end, reduces the advantage of the blocked method. After a square matrix side of 3000 (see Fig 9), the bandwidth consumed by the partial results negates the advantage of the reuse for matrix A.

D. Bandwidth Analysis

Looking back at our results, the effective bandwidth obtained from main memory is 1.3 word/cycle (theoretical 2GB/s, 500MHz clock, is 2 Word (4 bytes) per cycle). Given infinite bandwidth the clusters with 2 Multipliers and 3 Adders would be multiplier limited, achieving 16 multiplies per cycles (8 clusters). Given that in the naive approach, 2 words of main memory bandwidth are required when the matrices don’t fit inside the SRF, it takes 24 cycles for the memory system to provide 32 operands for the multipliers. The multipliers are thus idle 95% of the time. For the blocked approach, it depends on the number of blocks into which the matrices are divided,

VI. CONCLUSION

Our results show that, despite the potential for creative management of the SRF, conventional blocking yields better performance for matrix-matrix multiplication than a more intuitive approach. For very large matrices though the extra bandwidth consumed by the partial results overwhelms the advantage of the reuse of matrix A and the intuitive approach does better. This last result is not common to cache based architectures as far as the authors know. Because stream architecture do effective pre-fetching of the input data, they are sensitive to memory bandwidth, not latency. It might be that cache based architectures are more latency sensitive in matrix multiplication which would explain why the extra bandwidth of the blocked method for big matrices doesn’t appear.

In the future, our findings should be integrated into compilers that translate generic codes to the Imagine programming model taking into account the matrices sizes and the size of the SRF to choose the more appropriate method.

REFERENCES

Fig. 6. Kernel schedule for the naïve approach. While the multiply units (MUL0, MUL1) are heavily utilized, the kernel is still I/O-bound (the column on the right).

Fig. 7. Kernel schedule for the blocked approach, which is communication-limited (the right column).

Fig. 10. Software Pipelined Kernel execution (though mostly stalls) with memory transactions