Mapping Vector Codes to a Stream Processor (Imagine)

Mehdi Baradaran Tahoori and Paul Wang Lee

{mtahoori,paulwlee}@Stanford.edu

Abstract:

We examined some basic problems in mapping vector codes to stream processors. In particular, we studied two issues of mapping simple two-input operations to vector codes by running small vector code snippets on the Imagine programming infrastructure. Some conclusions from this study are as follows. Number of elements processed at one time, (record size from here on) has the same effect as unrolling and generally results in worse performance if pipelining is possible. Merging kernels to form larger kernels improve performance for most cases. There were some additional constraints to this, as is discussed in the relevant section.

1. Introduction

The large body of vector/vectorizable code in the scientific and engineering community presents great potential opportunities for stream architectures. Scientific codes often exhibit large degrees of data parallelism, and as such may be good candidates for SIMD streaming applications. Although there may be occasions where frequent data reuse and irregular accesses create problems for the stream programming paradigm, there will certainly be a large class of applications where stream architectures are more effective, due to the ability to take advantage of consumer-producer locality and thus make more efficient use of the memory bandwidth hierarchy.

For most applications, completely rewriting applications in a stream programming language would probably result in the best possible result. However, this is very costly and impractical. An automatic translation or mapping from legacy vector applications into stream language programs would be very useful in this respect.

This project explores some of the aspects of converting of vector codes into stream programs.

2. Overview

We had two avenues of approach in this project. The first was to scan through a representative set of vector applications and extract features that are favorable to efficient stream programs. The second approach was to experiment with short snippets of vector instructions on an existing stream programming system, namely the Imagine programming system, to find constraints and evaluate various simple mapping strategies. These two studies could be the foundation for classifying vector code structures in terms of their convertibility to streams, and devising efficient conversion methods for code that are convertible.

3. Vector applications

We scanned through a few scientific applications in FORTRAN, as well as one MATLAB code. The former included linear algebra packages and a small fluid dynamics program, and the latter some signal processing programs. Although the latter is not an exact match for our target area, much MATLAB code heavily use vectors and matrices and as such could yield the same insights.

These codes were not explicitly vectorized, and we quickly found that they were not very helpful for our purpose. This led to an investigation of a broader scope, of finding data parallel operations in programs. This would involve thorough dependency analysis of operations and data in the program. We found that extensive research already exists on the subject of vectorization of code, and there were a number of vectorizing compilers and translators. [Allen] discusses analyses and methods for translating non-vector FORTRAN to vector code. Vectorizing compilers may be used as a

preprocessing step on other legacy codes, or the analysis can be applied to directly obtain streaming structures. In the latter case, the analysis must be modified to take into account the restrictions as well as the additional capabilities of stream languages.

4. Experiments on Imagine

A number of experiments were performed on the Imagine programming system, which will be discussed in this section.

A. Partitioning

We considered two major types of partitioning of vector manipulations into kernel codes on Imagine. The partitioning can be done modulo data and modulo operation. In modulo data partitioning, the size of stream elements is changed as the parameter. For example, for a simple vector add, C[0:15] = A[0:15] + B[0:15], the kernel signature is as follows:

```
record vect {float v0, …, vn;}
kernel VADD(istream<vect> A, istream<vect> B,
            ostream<vect> C)
```

where n is the stream element size and the main parameter in modulo data partitioning.

The other dimension in partitioning is modulo operation. In this type of partitioning, the kernel granularity, the number of operations in each kernel, is considered. As an example, consider the following vector operations:

```
C[15:0] = A[15:0] + B[15:0]
E[15:0] = C[15:0] × D[15:0]
```

These operations can be coded in KernelC using two kernels, VADD(A,B,C) and VMUL(C,D,E), or one kernel, VADD_MUL(A,B,D,E).

We consider the effect of these two types of partitioning on the efficiency of the mapped code. All simulations except as noted in the last segment, were run on data set size (vector length) of 256 floating point elements.

B. Modulo Data Partitioning

In this section, we investigate the effect of stream element size, or vector size, on the execution time of mapped code. Figure 1 shows the effect of vector size on kernel scheduling. The representative vector operations are considered. To compare the results, the scheduling cycles are normalized to one data element. As can be seen in this figure, larger record sizes result in better scheduling.

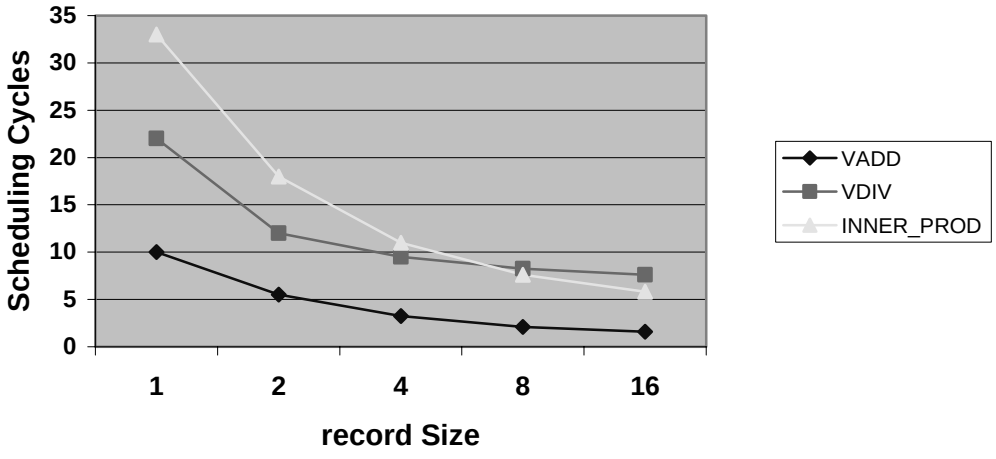

Figure 1 Scheduling normalized to one data element

As expected, by having a larger record size, more (independent) operations are exposed to the scheduler, increasing the resource utilization.

Figure 2 shows the total execution time for the kernels shown in Fig. 1. The results are not as expected, i.e. the total execution time increases as larger vector size is used.

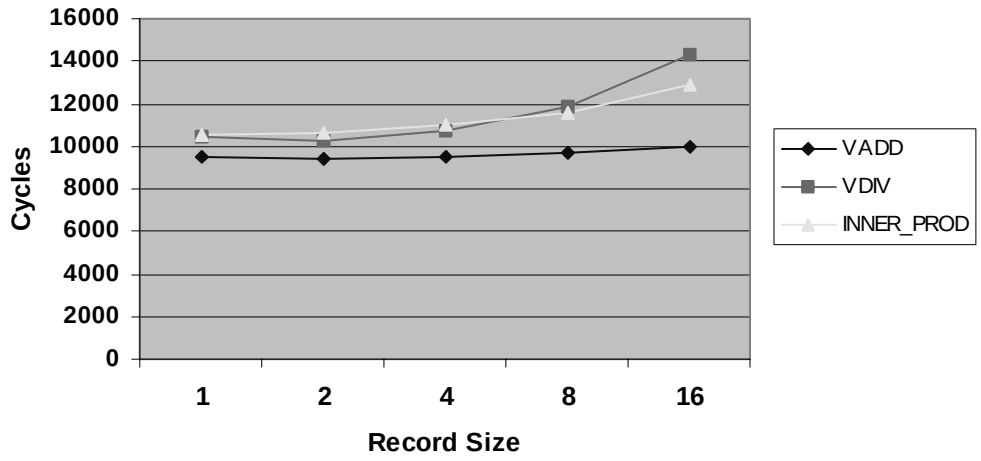

Figure 2 Total execution time

To explain the results, we looked at the detailed cycle count information form the simulator. The total execution time is broken down to time loading streams to SRF, loading micro-code, and kernel execution. Figure 3 shows the detailed cycle count for an add kernel. As the data set size is invariant (256 elements), stream load time is the same for all values of record size and hence it is not shown in this figure.

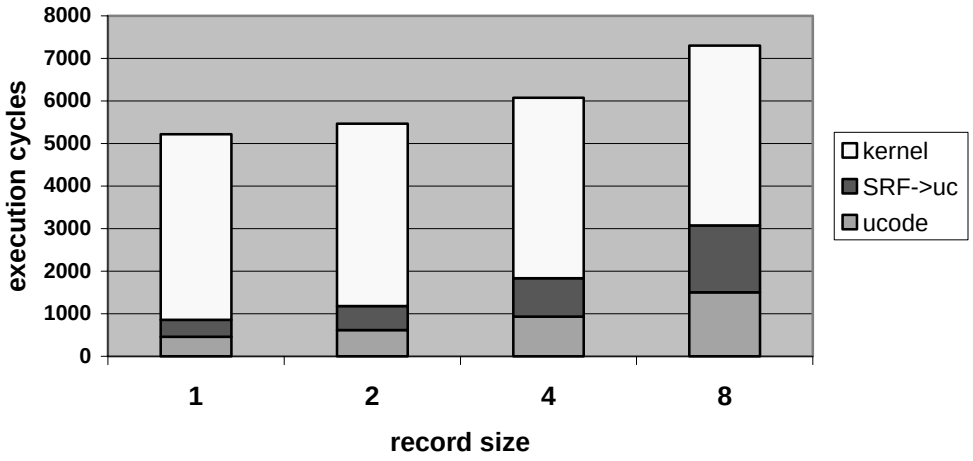

Figure 3 Detailed cycle count for ADD kernel

By using larger record sizes, µcode is also increased. This has the same effect of unrolling the code n times. Hence, µcode cost, consisting of loading µcode from main memory to SRF, and from SRF to µcontroller, is increased with larger record sizes, as can be seen in this figure. This is why the total execution time increases with larger record sizes despite of better scheduling of the kernel.

One solution to this problem is to amortize the µcode cost over several calling of the same kernel. In other words, if the kernels are small enough so that in the whole computation, the same kernel is called multiple times, the µcode cost of that kernel can be amortized over all the calls to that kernel. As an example, if we call the same ADD kernel 10 times in stream C code for performing 10 add operations, instead of having a big kernel for all the computations, the total execution time doesn't grow up with larger record sizes, as shown in Fig. 4.

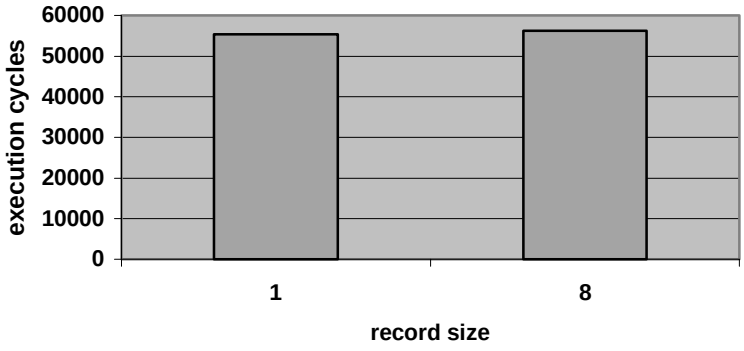

Figure 4 Total Execution time for 10 ADD kernels

Note that the µcode cost is significant only for short streams and/or computationally non-intensive kernels. For long enough streams, the µcode overheads are amortized over stream size.

The conclusion from this experiment is that for short streams, using small kernels in order to increase the possibility of reusing a kernel for other portions of computation, may enable amortization of μcode cost and improve performance.

C. Kernel Granularity

To study the effect of kernel granularity on execution time, we considered some simple vector manipulations. These codes along with their corresponding dataflow graphs are shown in Fig. 5. We considered two extreme cases, i.e. each operation in a separate kernel, and all operations in one kernel.

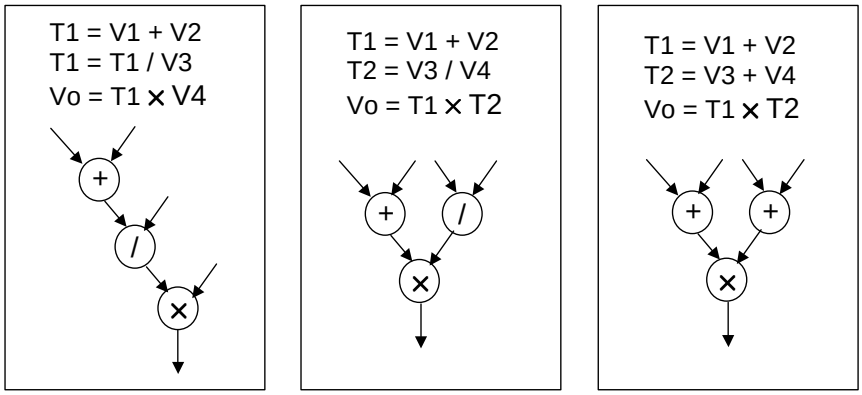

Figure 5 Some simple vector codes

The simulation results are shown in Fig 6. These results are on a data set of size 256 without performing software pipelining in kernel loops. As can be seen in this figure, for serial computations (Fig. 5.a), smaller kernels with smaller record size give the best performance. As the record size increases the μcode overheads of 3 kernels gets larger than those for one kernel, and hence the execution time of 3 kernels for larger record sizes would be larger. For non-serial (parallel) computations, bigger kernels are always

better. Exposing more operations to the schedulers, which can be executed in parallel, results in shorter execution time.

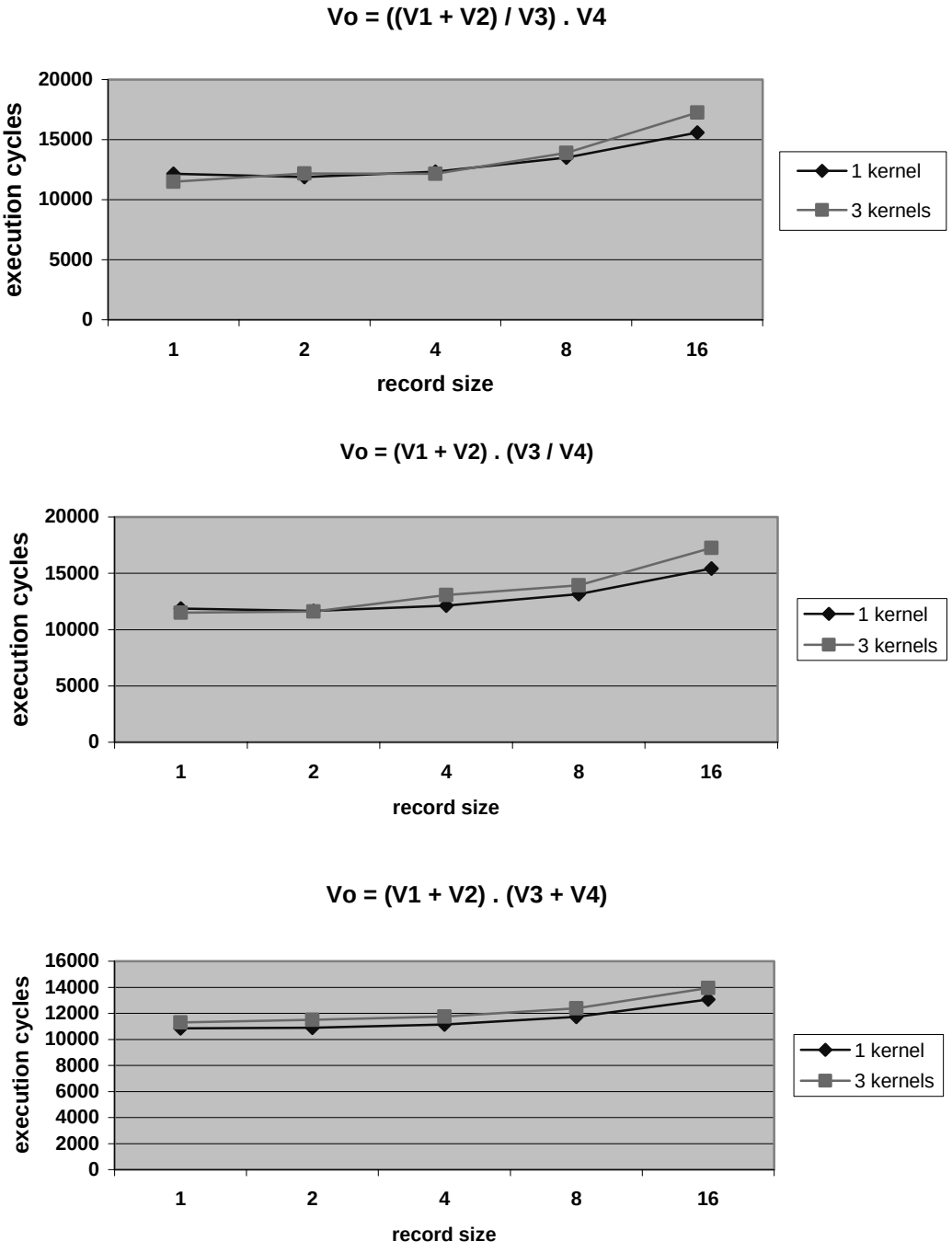

Figure 6 Simulation results for different kernel sizes

Computationally intensive vector operations were considered by performing simulations on codes with heavy loops. These carry independent loops, instances of which could be implementations of matrix by vector multiplications. The effect of software pipelining is considered on these codes. The simulations are performed on loops with 1000 iterations. The loop bodies are the code shown in Fig. 5. Figure 7 shows the simulation results.

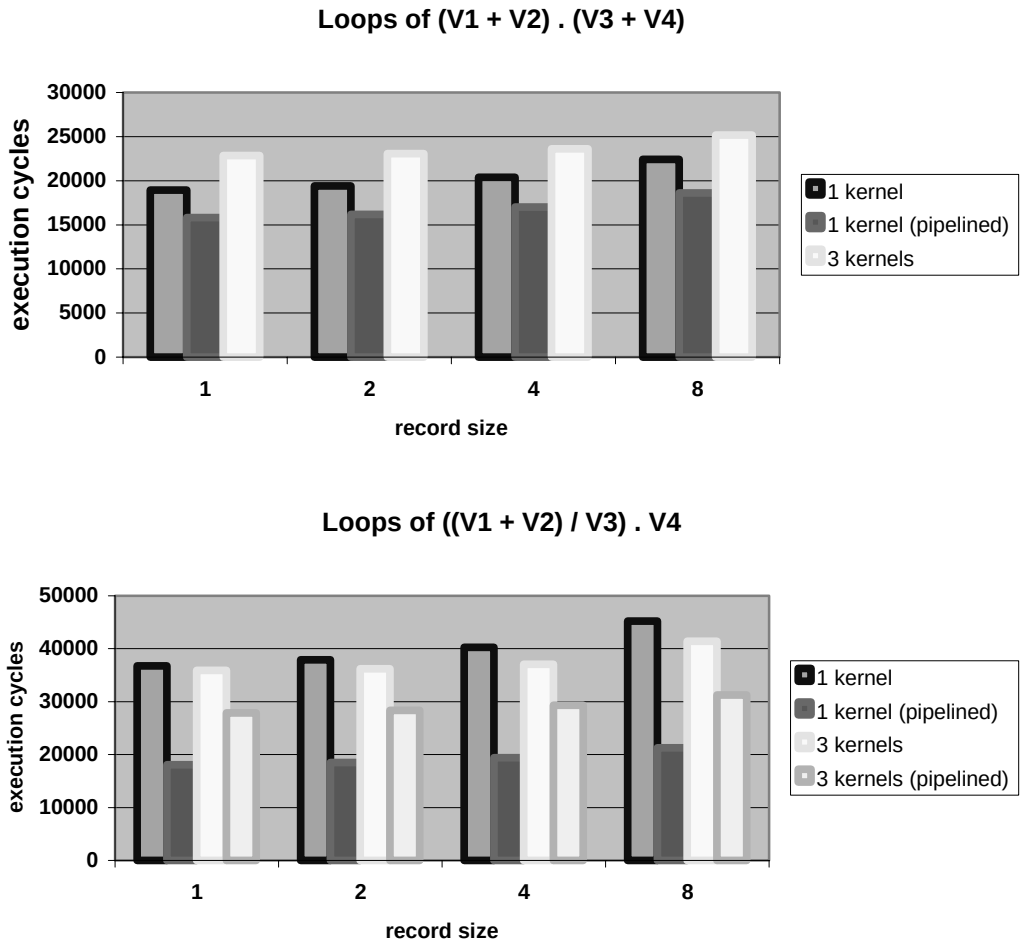

Figure 7 Simulation results for codes with loops and software pipelining

Again, for non-serial computations, having a big kernel results in shorter execution than several small kernels. For serial computations, without software pipelining, smaller kernels instead of one big kernel have better result.

By exploiting software pipelining, the total execution time reduces in all cases, but it gives more improvements with larger kernels. Hence, even for serial computations, bigger kernels results in better execution time by using software pipelining. Having more operations in one kernel gives more opportunities to the scheduler to pipeline the operations and generate more compact schedules with better resource utilization.

D. Larger data set studies

Additional studies were performed, for larger data sets and larger number of basic operations. Figure 8 shows the two instruction sequences where kernel grouping was performed. 4 different kernel groupings were run, from 1 operation per kernel to one large kernel. Data sets of 256, 2048, 4096 and 8192 were run for each of the above 8 configuration/grouping pairs. These were run on the cycle accurate simulator, with the profiling compiler. Due to the limit on the maximum number of streams a kernel can have, the actual number of operations was limited to 6, for 7 input streams. For a comparison to be possible between the two sequences, 6 input streams and 5 operations were chosen for both. The results are shown in figures 9 and 10. The y-axis is normalized run time, i.e. cycle time taken to process each data value.

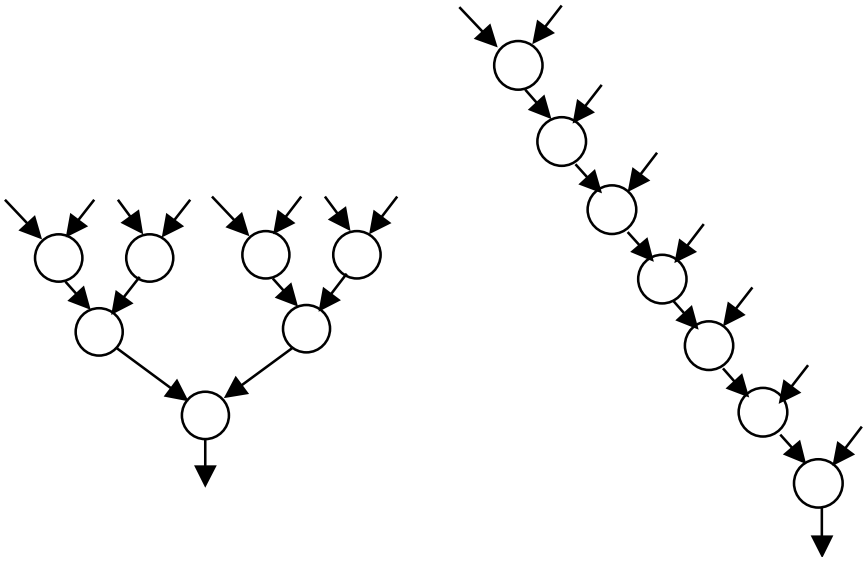

Figure 8 Parallel and serial chains of operations

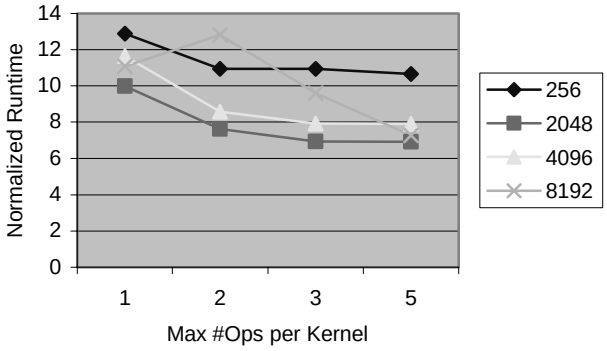

Figure 9 Serial instruction chain for various kernel groupings

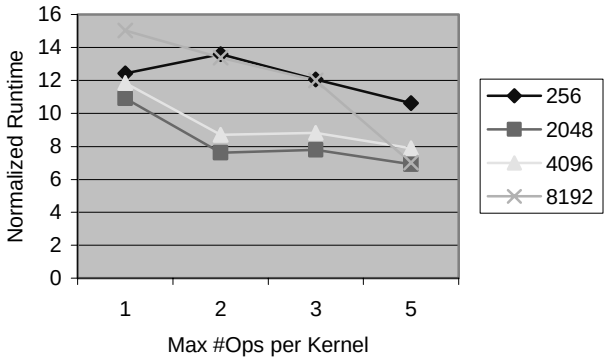

Figure 10 Parallel chain with various kernel groupings

The above results show the following trend. In general, larger kernels perform better, due to better scheduling within the kernels, and possibly smaller total kernel startup overhead. However, once a sufficient number of operations are scheduled in the kernel, the added benefit becomes small. For our simulations, the operations were floating point adds, and after two, which is the number of floating point adders per arithmetic cluster in the simulated machine, the settling down is evident. The reason for using only floating point adds in our instruction sequences was to isolate effects from different latency operations, and also to stress the resources in the functional units. The graphs also show a consistent trend of better normalized performance for larger data sets. This is a result of amortization of fixed costs. The data size of 8192 bucks the trend and becomes larger, but this is due to the fact that this

length stresses the SRF and the streams start to become double buffered and spilled. Once the streams start spilling, any reduction in number of kernels is better, since this essentially eliminates an intermediate stream by storing in the LRF and reusing immediately in the kernel. There are two features of note on the graph. In figure 10, the 256 data size case runs faster for single-op kernels. This is due to the explanation given in previous sections, namely the microcode transfer time. If the operations were all different for each of the separate 1-op kernels, this effect would disappear and its time would be larger. 8192 data sets in Figure 9 is more interesting. It seems that since the number of streams required to run a single kernel is fewer for 1-op kernels, (2 input streams vs. 3 input streams for 2-op kernels) the stream register file can keep fewer values live and thus make more efficient use of the existing space, thereby reducing spilling. This is interesting, and also explains the fact that the serial sequence usually runs faster than a parallel sequence. However, with stripmining this effect would go away. From these sets of experiments, one can conclude that grouping sequences of kernels together according to the hardware resource mix would result in best performance. There are two additional considerations. Namely, the number of streams are limited for a kernel, and this would limit the number of operations which need new inputs, and would act as a constraint under which optimal grouping should be performed. Also, in our experiments, the size of LRF's were not a limiting factor. Intuitively, keeping more data that can be used immediately in the LRF's would reduce SRF demand and improve performance in many cases. However, since LRF's are also a limited resource, how to group kernels such as to make the most effective use of available LRF space would be an important problem for further study. Also, it is worth noting that the problem of grouping single vector operation to large kernels is a specific case of kernel fusion and general results in this area may apply to this case.

5. Conclusions

We have explored some aspects of mapping vector codes to stream codes. A look at legacy codes and vectorizing compiler research provide base material for converting legacy codes in general to stream programs. We performed simple experiments in the Imagine system and identified some of the major considerations in the transformation of vector codes to stream programs, such as record and kernel granularity and the corresponding constraints. Suggestions for further research include additional studies for more non-straightforward operations that involve reduction and communication, developing a set of heuristics for optimum kernel fusion, and composing a general classification of vector code structures and the corresponding mappings that can be used for automating translation from vector to stream codes.

6. References

[Allen] Randy Allen, Ken Kennedy "Automatic Translation of FORTRAN Programs to Vector Form," *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 4, October 1987, Pages 491-542.