Journal of Computational Science 3 (2012) 120–126

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Modeling the memory and performance impacts of loop fusion

Ian Karlin*, Elizabeth Jessup, Erik Silkensen

University of Colorado at Boulder, Department of Computer Science, 430 UCB, Boulder, CO 80309, United States

ARTICLE INFO

Article history:
Received 8 October 2010
Received in revised form 21 January 2011
Accepted 2 March 2011
Available online 16 March 2011

Keywords:
Performance tuning
Memory-bound computation
Linear algebra
Memory modeling

ABSTRACT

On modern processors, data transfer exceeds floating-point operations as the predominant cost in many linear algebra computations. One tuning technique that focuses on reducing memory accesses is loop fusion. Determining the optimum amount of loop fusion to apply to a routine is difficult as fusion can both positively and negatively impact memory traffic. We present a model that accurately and efficiently evaluates how loop fusion choices affect data movement through the memory hierarchy. We show how to convert the model's memory traffic predictions to runtime estimates that can be used to compare loop fusion variants.

1. Introduction

Historically, the performance of tuned linear algebra computations has been measured in terms of floating-point operations per second (flops). For operations like matrix–matrix multiplication that perform many flops per memory access, such a measurement can still be valid [1]. However, for other calculations, performance can be limited by the cost of passing data from memory to the processor [2]. The cost of these memory-bound operations is more accurately expressed in terms of reads and writes than flops.

Recently, the focus of tuning linear algebra operations has shifted to minimizing memory traffic, often resulting in speedups equivalent to its reduction [3,4]. This optimization can increase program speed even if the number of flops performed increases as a result [5]. One technique to lessen data movement is to combine or fuse the loops of multiple calculations [6]. Loop fusion has been applied successfully to single kernels [7] and as well as to such important larger scale operations as Householder bidiagonalization where speedups of 10–25% have been reported [4].

To aid in the implementation of arbitrary sequences of loops, compilers are being improved to include loop fusion as a possible optimization [8,9]. To find efficient routines, these compilers employ various methods to figure out how much fusion to apply. However, finding all possible ways to fuse loops of general computations is NP-complete [10], and, therefore, testing all possibilities to determine the fastest can become expensive for even small problems. Thus, in some cases, heuristics govern the optimizations [9]. In others, search is used to figure out the best routine [11]. Additionally, search can be combined with modeling [12] to direct a compiler on which fusion decisions are appropriate. However, both heuristics and guided search can produce suboptimal routines when they do not evaluate all fusion options. In particular, directed search can identify local extrema [13].

As we show in [14], an enumeration of the entire search space for a given routine is often feasible in the restricted domain of linear algebra. There, we also introduce a model that dramatically reduces search time. In this paper, we explain how our loop fusion research directs the selection of routine and hardware features included in the model. By not including the entire machine, we are able to significantly decrease our model's runtime at a small accuracy cost. Our model builds on the work of others to predict memory misses [15], turn data access pattern information into runtime predictions [16] and use data movement to direct search and find high performing loop fusion routines [12]. It uses a unique set of tradeoffs designed to balance runtime and accuracy for a compiler that enumerates all possible ways to fuse a linear algebra routine. Using the model, we are able to accurately and efficiently compare multiple variants of the same linear algebra routine with differing amounts of fusion, thus reducing the amount of time it takes the compiler to generate efficient fused kernels.

The rest of the paper is organized as follows. In Section 2, we present examples that show how loop fusion can increase or decrease memory traffic to motivate why modeling is important to producing efficient routines. In Section 3, we describe how to generate accurate memory estimates for fused linear algebra routines, which we encode into the model. In Section 4, we explain how we convert those estimates to runtime predictions, which are

* Corresponding author.
E-mail address: Ian.Karlin@colorado.edu (I. Karlin).

1877-7503/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.jocs.2011.03.002

then used to compare routines. Conclusions and future work are discussed in Section 5.

2. How fusion impacts data movement

When successful, loop fusion reduces the amount of data that must move through the memory hierarchy by increasing the locality of data accesses. The memory hierarchy, which stores and moves data to perform calculations, includes registers, caches, translation lookaside buffers (TLBs) and main memory. Registers store data immediately accessible by the processor, while caches store a subset of the data from main memory physically closer to the processor than the memory is. TLBs are used to speed the virtual to physical translation of memory addresses that is necessary on processors using virtual memory, which almost all modern processors do. Caches are arranged into lines, which typically hold 32–128 bytes of data that are stored consecutively in memory. Each entry in a TLB stores the translation for one page of main memory. A page is usually 4 kB in size.

For memory-bound operations, loop fusion can increase performance nearly proportionately to the reduction in memory traffic. However, too much fusion can negatively impact performance by causing register spill, which occurs when the compiler cannot allocate a register for a datum, as well as TLB and cache misses. In this section, we show how fusion affects data movement through the memory hierarchy and the consequences for routine performance.

In order to observe effects that occur when many loops are fused, we look at what happens when a number of matrix–vector multiplications are combined. We focus on large matrix orders in our experiments because small matrix problems often fit within cache and are computationally bound. To begin, we define a routine DGEMV2 that multiplies vectors u_0 and u_1 in turn by a matrix A as shown in Fig. 1. Fig. 2 shows the three possible ways to fuse the routine: no loop fusion, only outer loops fused, and all loops fused.

```
DGEMV2
in
  u0 : vector, u1 : vector,
  A : row matrix
out
  v0 : vector, v1 : vector
{
  v0 = A * u0
  v1 = A * u1
}
```

Fig. 1. DGEMV2.

Table 1
Specifications of the test machines. For TLBs, we list the number of entries. On the Opteron there are two and the number of entries is listed for each.

Processor	Speed	Mem	Bus speed	L1	L2	TLB
Intel Core 2	2.4 GHz	2 GB	1333 MHz	32 kB	4 MB	256
AMD Opteron	2.6 GHz	3 GB	1000 MHz	64 kB	1 MB	40/512

2.1. Fusion improves performance

Fusing the two matrix–vector multiplications leads to better memory usage and so better performance. Fig. 3 shows that, on the Opteron system described in Table 1, fusing the outer loops improves the megaflop rating by 60% over the unfused routine and fusing all loops increases it by 70%. The figure also shows that both fused routines reduce L2 cache misses by approximately one half. TLB misses are also reduced by one half for both routines. However, L1 cache misses and executed load instructions are only reduced by one quarter for the fully fused routine. For the outer loop fused routine, L1 cache misses and executed load instructions are nearly identical to those of the unfused routine. These experiments and all others presented in this paper were compiled using the Intel compiler icc with the −O3 flag. All matrices tested were dense.

2.2. Fusion degrades performance

Loop fusion does not always result in improved performance. When fusion requires more data to fit in a memory structure of the computer than the structure can hold, memory traffic increases [17,18]. To explore how loop fusion can increase data movement through the caches and registers, we expand the previous experiment to the fusion of an arbitrary number *nvecs* of matrix vector multiplications Au_x, $x = 1, \ldots,$ *nvecs*.

2.2.1. Cache effects

When the outer loops of the *nvecs* matrix–vector multiplications are fused, performance dropoffs occur as shown in Fig. 4. In the fused routine, all operand vectors u_x, $x = 1, \ldots,$ *nvecs*, are accessed during the iterations of the single outer loop. For large matrix orders, the combined size of the vectors is larger than cache meaning that they must be read from memory. That increase in L2 cache misses is demonstrated in Fig. 4. When all loops are fused cache misses occur as well, but in order to isolate the impact of the cache from registers we show results using outer loop fusion.

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    v0[i] += A[i][j] * u0[j]
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    v1[i] += A[i][j] * u1[j]
```

(a) No Fusion

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    v0[i] += A[i][j] * u0[j]
  for (j = 0; j < n; j++)
    v1[i] += A[i][j] * u1[j]
```

(b) All Outer Loops Fused

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    v0[i] += A[i][j] * u0[j]
    v1[i] += A[i][j] * u1[j]
```

(c) All Loops Fused

Fig. 2. Three possible loop fusion options. (a) No fusion, (b) all outer loops fused, and (c) all loops fused.