HIGH-LEVEL LANGUAGE EXTENSIONS FOR FAST EXECUTION OF PIPELINE-PARALLELIZED CODE ON CURRENT CHIP MULTI-PROCESSOR SYSTEMS

Apan Qasem

1Department of Computer Science, Texas State University San Marcos, Texas, USA
apan@txstate.edu

ABSTRACT

The last few years have seen multicore architectures emerge as the defining technology shaping the future of high-performance computing. Although multicore architectures present tremendous performance potential, to realize the true potential of these systems, software needs to play a key role. In particular, high-level language abstractions and the compiler and the operating system should be able to exploit the on-chip parallelism and utilize underlying hardware resources on these emerging platforms. This paper presents a set of high-level abstractions that allow the programmer to specify, at the source-code level, a variety of parameters related to parallelism and inter-thread data locality. These abstractions are implemented as extensions to both C and Fortran. We present the syntax of these directives and also discuss their implementation in the context of source-to-source transformation framework and autotuning system. The abstractions are particularly applicable to pipeline parallelized code. We demonstrate the effectiveness of these strategies of a set of pipeline parallel benchmarks on three different multicore platforms.

KEYWORDS

Language abstractions, parallelism, multicore architecture

1. INTRODUCTION

The last few years have seen multicore and manycore systems emerge as the defining technology shaping the future of high-performance computing. As the number of cores per socket grows, so too does the performance potential of these systems. However, much of the responsibility of exploiting the on-chip parallelism and utilizing underlying hardware resources on these emerging platforms lies with software. To harness the full potential of these systems there is a need for improved compiler techniques for automatic parallelization, managed runtime systems and improved operating systems strategies. Most importantly there is a need for better language abstractions that allow a programmer to express parallelization in the source code and specify parameters for parallel execution. Concomitant to these abstractions, there is a need for software that is able to translate the programmer directives to efficient parallel code across platforms.

This paper presents a set of high-level language abstractions and extensions that allow the programmer to specify, at the source-code level, a variety of parameters related to parallelism and inter-thread data locality. These abstractions are implemented as extensions to both C and Fortran. We present the syntax of these directives and also discuss their implementation in the context of a source-to-source transformation framework and autotuning system. These abstractions are particularly suitable for pipeline-parallelized code. Multicore architectures, because of their high-speed inter-core communication have increased the applicability of...
pipeline-parallelized applications. Thus, the abstractions described in this paper are especially significant. We demonstrate the effectiveness of these strategies on a set of benchmarks on three different multicore platforms.

The rest of the paper is organized as follows: in Section 2, we present related work; in Section 3 we describe the language extensions; the tuning framework is described in Section 4; experimental results appear in Section 5 and finally we conclude in Section 6.

2. RELATED WORK

The code transformations described in this paper have been widely studied in the literature. Tiling has been the predominant transformation for exploiting temporal locality for numerical kernels [20, 5, 3]. Loop fusion and array contraction have been used in conjunction for improving cache behavior and reducing storage requirements [4, 13]. Loop alignment has been used as an enabling transformation with loop fusion and scalarization [2]. The iteration-space splicing algorithm used in our strategy was first introduced by Jin et al. [6]. Time skewing has been a key transformation for optimizing time-step loops. McCalpin and Wonnacott introduced the concept in [10] and Wonnacott later extended this method for multiprocessor systems [23]. Wonnacott’s method imposed restrictions on dependencies carried by spatial loops and hence was not applicable for general time-step loops. Song and Li describe a strategy that combines time skewing with tiling and array contraction to reduce memory traffic by improving temporal data reuse [13]. Jin et al. describe recursive prismatic time skewing that perform skewing in multiple dimensions and handles shadow regions through bi-directional skewing [6]. Wolfe introduced loop skewing, a key transformation for achieving pipelined parallelism [21]. Previously, pipelined parallelism was mainly limited to vector and systolic array machines. However, with the advent of multicore platforms there has been renewed interest for exploiting pipelined parallelism for a larger class of programs. Because of their regular dependence patterns, much of the attention has been centered on streaming applications [15, 17, 8]. Thies et al. [14] describe a method for exploiting coarse-grain pipelined parallelism in C programs. Vadlamani and Jenks [18] present the synchronized pipelined parallelism model for producer-consumer applications. Although their model attempts to exploit locality between the producer and consumer threads they do not provide a method for choosing the appropriate synchronization interval (i.e. tile size).
3. LANGUAGE ABSTRACTIONS

We propose several language extensions in C and Fortran that allow a programmer to express constraints on parallelism and also provide directives and suggestions to the compiler (or a source-to-source code transformation tool) about applying certain optimizations that improve the performance of the code. The syntax for these extensions follows closely the syntax used in OpenMP programs, increasing the usability of the proposed strategy. Fig. 2(a) shows an example of directives embedded in Fortran source code. A directive is simply a comment line that specifies a particular transformation and one or more optional parameter values, which can be associated with any function, loop, or data structure. The association of directives with specific loops and data-structures is particularly useful because current state-of-the-art commercial and open-source compilers do not provide this level of control. For example, GCC does not allow the user to specify a tile size as a command-line option whereas Intel's icc allows a user-specified tile size, but applies it to every loop nest in the compilation unit. The use of source-level directives in provides a novel and useful way of specifying optimization parameters at loop-level granularity, enhancing the effectiveness of both automatic and semi-automatic tuning more effective.

Each source directive is processed by CREST, a transformation tool which serves as a meta compiler in our framework. Each directive is inspected, the legality of code transformation is verified and profitability threshold is determined. In the autotuning mode, the parameters are exposed for tuning through heuristic search whereas in the semi-automatic mode, the performance feedback is presented to the programmer. The rest of this section describes the four main directives used in optimizing pipelined parallelized code.

For example, tile size and shape can be used to determine the number of required synchronizations and the amount of work done per thread. Some of the transformations currently supported by CREST include tiling, unroll-and-jam, multi-level loop fusion and distribution, array contraction, array padding, and iteration space splicing.

3.1. Skew Factor

A key consideration for parallel pipeline code is the amount of time spent in the filling and draining as a fraction of the time spent in the steady state. To address this issue we include an language extension that allows the user to specify a skew factor that determines the size of the fill-and-drain time in the pipelined parallelized code. The skew factor directive takes a single integer parameter as shown in Fig. 2. The iteration space depicted in Fig. 2 has carried dependencies in both the inner and outer loops. Loop skewing can be used to remap the iteration space and expose the parallelism along the diagonals of the original iteration space, as shown in
Fig. 1(b). Skewing results in a trapezoidal iteration space where the triangular regions represent the pipeline fill-and-drain times and the rectangular region in the middle represents the steady state. In a skewed iteration space, the area of the triangular regions is determined by the size of the x dimension (the outer loop). If we assume

\[ n = \text{loop bounds for outer loop} \]
\[ ov = \text{pipeline fill-and-drain time} \]
\[ s = \text{pipeline steady state} \]

Then, we get

\[ ov = 2 \times \frac{1}{2} x (n - 1) \times (n - 1) = (n - 1)^2 \]

Therefore, \( ov \) will increase and dominate execution time for larger time dimensions (this scenario is depicted in Fig. 2(a)). This can cause severe performance degradation because of high synchronization overhead. To alleviate this situation, we propose blocking along the time dimension to produce a blocked iteration space as depicted in Fig. 2(b). This scheme reduces the

\[
\begin{align*}
\text{do} & \ i = 1, M, 64 \\
\text{do} & \ j = 1, N \\
\text{do} & \ ii = i, i + 64 - 1 \\
& b(ii, j) = a(ii, j) + a(ii, j-1) \\
& b(i, j) = a(i, j) + a(i, j-1) \\
& c(ii, j) = b(ii, j) + d(j) \\
& c(i, j) = b(i, j) + d(j) \\
\text{enddo} \\
\text{enddo} \\
\text{enddo}
\end{align*}
\]

second, since each iteration of the outer loop performs roughly the same amount of work, the maximum parallelism is bounded by the available parallelism on the target platform. Therefore, by picking \( B_H \) to be the amount of available parallelism, we ensure that we do not lose any parallelism as a result of blocking. For this work, we assume available parallelism to be the number of cores (or HW threads if hyperthreading is present) available per node.

3.2. Data Locality Enhancement

Exploiting data reuse continues to be a key consideration for achieving improved performance. In particular, on current multicore architectures with shared-cache it is of paramount importance that in addition to exploiting intra-thread locality we focus on inter-thread locality as well. We

propose several language extensions that allow a program to specify optimizations that can be applied by a tool at the source-code level. These transformations include: loop interchange, unroll-and-jam, loop fusion, loop distribution, iteration space splicing and tiling. Each of these optimizations can be applied to affect changes in the both inter and intra-thread data reuse. Further the transformation can also be applied to modify thread granularity, which can also have a significant impact on the performance of pipeline-parallelized code. Fig. 3(a) shows directives for tiling in Fortran source. Fig 3(b) shows the resulting code after the transformation has been by CREST.

The principal technique that we apply to exploit data reuse in pipelined parallelized code is multi-level blocking. For an n-dimensional loop nest, the loop at level n – 1, is tiled with respect to the loop at level n (outermost loop). The main issue here of course is picking the right tile size. Because the tile size controls not only the cache footprint but also the granularity of parallelism, we need to be especially careful in selecting a tile size. The ideal tile size is one that gives the maximum granularity without overburdening the cache. To find a suitable tile size for a shared cache, we first determine the cache footprint for all co-running thread. This information is obtained through reuse analysis of memory references in the loop nest in conjunction with thread scheduling information. According to our scheduling strategy, the data touched by a block $b_i$ is reused $d$ steps after the completion of $b_i$, where $d$ is the block coverage value as derived in the previous section. Our schedule allows for $p$ concurrent threads at each stage. Hence, to exploit temporal locality from one time step to

the next, we pick a block size $B$ that satisfies the following constraint

$$(d+1) (f_{b_0} + f_{b_1} + \ldots f_{b_p}) < CS$$

(1)

where $f_{bi} =$ cache footprint of thread $i$, $CS =$ estimated cache size

For many applications however, the above condition might be too relaxed. We can enforce a tighter (and more profitable) constraint by considering the amount of data shared among concurrent threads. Generally, our synchronization-free schedule implies we have no true dependencies between concurrent threads. However, it is common to have input dependencies from one block to another. We do not want to double count this shared data when computing the cache footprint of each thread. Thus, we can revise (1) as follows

$$(d + 1) \times (fb_0 + fb_1 - sh(b_0-b_1) + fb_2 - (sh(b_0-b_2) + sh(b_1-b_2)) \ldots + fb_p - (sh(b_0-bp) + \ldots sh(bp-1-bp))) < CS$$

(2)

where $sh(b_1-b_2) =$ amount of data shared among between $b_1$ and $b_2$

There is one other aspect of data locality that needs to be considered when choosing a tile size. We need to ensure that our tile size is also able to exploit intra-block locality. If we were running the loop nest sequentially then a tiling of the inner spatial dimensions would suffice. However, for the parallel case we need to tweak the outermost tile size to exploit the locality within the inner loops. In particular, we need to ensure that the concurrent threads do not interfere with each other in cache. The constraint on the tile size we enforce with (8) handles this situation for the last-level cache. Hence, we add a secondary constraint on the tile size that targets higher-level caches for exploiting intra-block locality.
3.3. Avoiding Shadow Regions With Thread Affinity

Shadow regions or ghost cells is another that needs to be considered when orchestrating pipelined parallel code. If access to shadow regions causes data from non-local memory to be fetched then it can cause significant performance degradation. To address this issue, we propose two high-level language extensions that can be used to specify the schedule and affinity of threads. The scheduling distance is specified using a single integer value whereas the affinity information is specified using a sequence of integers as shown in Fig. 4.

We propose an alternate schedule to avoid this synchronization overhead. In our schedule, we distance the concurrent threads \( d \) blocks apart, where \( d \) represents the block coverage of the shadow region in terms of number of blocks. Of course, in most cases the value of \( d = 1 \). Scheduling the threads \( d \) blocks apart creates a safe zone for each block to execute without having to synchronize with any of the blocks from the previous time-step. Of course, in this schedule the fill-and-drain time is increased. Even for a dependence distance of 1, the fill-and-drain overhead will be twice as much. However, this overhead is significantly less than the overhead of synchronization for shadow regions.

4. FRAMEWORK OVERVIEW

Fig. 1 provides an overview of our transformation framework. The system comprises of four major components: a source-to-source code restructurer (CREST), a program analyzer and feature extractor, a set of performance measurement tools, and a search engine for exploring the optimization search space. MATS can operate in two modes: offline and online. In the online mode, the source code is first fed into the program analyzer, which utilizes program characteristics and architectural profiles gathered at install time to generate a representative search space. The search engine is then invoked, which selects the next search point to be evaluated. Depending on the configuration, the search point gets translated to a set of parameters for high-level code optimizations or compiler flags for low-level optimizations or a combination of both. The program is then transformed and compiled with the specified optimizations and executed on the target platform. During program execution, performance measurement tools collect a variety of measurements to feed back to the search module. The search module uses these metrics in combination with results from previous passes to generate the next set of tuning
parameters. This process continues until some pre-specified optimization time limit is reached or the search algorithm converges to a local minima. In the offline mode, the source program is fed into the feature extractor, which extracts the relevant features and then maps it to one of the pre-trained predictive models using a nearest neighbor algorithm. The predictive model emits the best optimization for the program based on its training. The gathering of training data and the training of the predictive models occurs during install time. Next, we describe the four major components of our adaptive tuning system and highlight their key features.

4.2. Program Analyzer and Feature Extractor

When a source program arrives to the framework, it is initially dispatched to a tool for program analysis and feature extraction. The program analyzer uses a dependence-based framework to determine the safety of all high-level code transformations that may be applied at a later phase by the code restructurer. The analyzer incorporates models that aim to determine the profitability of each optimization and based on these models, it creates the initial search space. The analyzer also contains the analysis needed to generate the pruned search space of architectural parameters, as explained in Section 7. Furthermore, at this stage, hot code segments are identified to focus the search to code regions that dominate the total execution time. The feature extraction module builds and analyzes several program representations including data and control-flow graphs, SSA and def-use chains. A variety of information that characterizes a program's behavior is extracted from these structures. Table 1 lists some example features collected by our tool. A total of 74 different features are collected for each program. The extracted features are encoded in an integer vector and stored in a database for use by the machine learning algorithms.

4.3. Source-to-source Code Transformer

As mentioned in Section 3, our code-restructuring tool, CREST, provides support for the language abstractions and is capable of performing a range of complex loop transformations to improve data reuse at various levels of the memory hierarchy. One attribute that distinguishes CREST from other restructuring tools is that it allows retargeting of memory hierarchy transformations to control thread granularity and affinity for parallel applications.

4.4. Feedback Parser and Synthesizer

MATS utilizes HPCToolkit [36] and PAPI [37] to probe hardware performance counters and collect a wide range of performance metrics during each run of the of the input program. Measurements collected in addition to the program execution time include number of cache misses at different levels, TLB misses and number of stalled cycles. HPCToolkit supports measuring performance of fully optimized executables generated by vendor compilers and then correlating these measurements with program structures such as procedures, loops and statements. We leverage HPCToolkit's ability to provide _ne-grain feedback with CREST's _ne-grain control over transformations to create search instances that can explore independent code regions of an application, concurrently. To use HPCToolkit in MATS, we developed FeedSynth, a feedback parser and synthesizer, which parses the performance output produced by HPCToolkit and delivers it to the search engine.

4.4. Search Engine

Our framework supports both offline and online tuning. The search engine implements a number of online search algorithms including genetic algorithm, direct search [38], window search, taboo search, simulated annealing [39] and random search. Apart from random search, all other algorithms are multidimensional in nature, which facilitates non-orthogonal exploration of the
For offline tuning, we implement several supervised and unsupervised learning algorithms. The implemented algorithms include support vector machines (SVM), k-nearest neighbor, k-means clustering and principal component analysis (PCA). Additionally, to support these learning algorithms, we implement three statistical predictive models including independent and identically distributed model (IID), Markov chain and logistic regression.

Table 1. Evaluation platforms

<table>
<thead>
<tr>
<th></th>
<th>Core2</th>
<th>Quad</th>
<th>8Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>No of cores</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Processor</td>
<td>2.33 GHz</td>
<td>2.4 GHz</td>
<td>2.53 GHz</td>
</tr>
<tr>
<td>L1</td>
<td>16 KB, 4-way</td>
<td>32 KB, 8-way</td>
<td>32 KB, 8-way</td>
</tr>
<tr>
<td>L2</td>
<td>4 MB, 4-way</td>
<td>2 x 4 MB, 8-way</td>
<td>8 MB, 16-way</td>
</tr>
<tr>
<td>Compiler</td>
<td>GCC 4.3.2</td>
<td>GCC 4.3.2</td>
<td>GCC 4.3.2</td>
</tr>
<tr>
<td>OS</td>
<td>Linux</td>
<td>Linux</td>
<td>Linux</td>
</tr>
</tbody>
</table>

5. EVALUATION

To evaluate the effectiveness of the ideas we conducted a series of experiments where we utilized the language extensions two three Fortran applications: a multigrid solver (multigrid), an optimization problem (knapsack), and an advection code excerpted from the NCOMMAS [19] weather modeling application (adve3d). All three applications are amenable to pipelined parallelism, and hence serve as suitable candidate for evaluating our strategy.

Each parallel variant is evaluated on three different multicore platforms. Table 1 shows the hardware and software configurations of each experimental platform. The two-core system is based on Intel's Conroe chip, in which the L2 cache is shared between the two cores. The four-core system uses Intel's Kentsfield chip, which has two L2 caches, shared between two cores on each socket. The eight-core platform has a Xeon processor with Hyperthreading (HT) enabled, providing 16 logical cores. The rest of this paper refers to the two- four- and eight-core platforms as Core2, Quad and 8Core. In addition to these three platforms, we also conduct scalability studies on a Linux cluster. The configurations for this platform are given in Section 5.3.

Figure 5. Performance sensitivity to time skew factors on Quad and 8Core.
5.1. Effectiveness of Time-Skew Abstraction

We first conduct a set of experiments to determine how variations in the time-skew factor influence the overall performance of the two applications. Fig. 5 shows performance sensitivity of knapsack and advect3d on Quad and 8Core, respectively. In the figure, speedup numbers are reported over a baseline version in which pipelined-parallelism is applied but no blocking is performed.

We observe that changes in the skew factor do indeed influence the performance of both knapsack and advect3d significantly. We see integer factor deviations in performance on both platforms. Although no direct relationship between skew factors and performance is evident in these two charts, there are two performance trends that are noteworthy. First, we note that small skew factors are generally not profitable. In fact, for factors less than 8, performance is worse than baseline in most cases. The second trend that we notice is that beyond a certain factor the performance curve tends to fatten out. This occurs, at factors 52 and 44 on Quad while for 8Core the performance plateaus arrive at 56 and 64, respectively. These performance trends suggest that a compiler-level heuristic may be able to search through a range of skew factors, where very small values and very large values are eliminated, a priori. We intend to incorporate such pruning strategies (in conjunction with autotuning) in our framework in future. The main take-away from these experiments is that skew factors can cause significant performance fluctuations and therefore, it is imperative that some language support is provided for applying this transformation.

5.2. Memory Performance

Since the proposed language extensions aim to improve data locality in cache, in addition to reducing synchronization cost, we conduct a set of experiments where we isolate the memory performance effects. Fig. 6 shows the normalized Level 2 cache misses for different blocking schemes for knapsack and advect3d, respectively. We focus on L2 because it is a shared-cache on all three platforms and our scheme specifically targets shared-cache locality. The numbers reported are for three different tiling schemes: outer loop only, inner loop only and both loop with skewing. In all three instances, the block size is the one recommended by our model. We observe that except for one case (advect3d on Quad), our heuristic is able to pick block and skew factors that have a positive impact on L2 performance. We also notice that generally the benefits gained from inner loop blocking is less than those obtained from blocking of the outer loop. Moreover, when applied together the blocking factors interact favorably and the best L2 figures are seen when tiling is performed for both dimensions. The strategy has is similarly effective on advect3d and knapsack. Also, tiling is most effective on 8Core. This is attributed to the smaller caches on this platform. Because of the 1 MB L2 on the 8Core (as

opposed 2 MB), there were more cache misses in the unblocked variants, which were eliminated through blocking.

![Performance Chart](chart.png)

5.3. Performance Scalability

We conducted experiments to evaluate performance scalability with respect to number of cores. To test the effectiveness of our strategy as the number of cores is increased, we implemented MPI variants of all three applications and executed them on a Linux cluster with 128 cores on 32 nodes. We ran multiple variants of each code, using the job scheduler to limit the number of cores that is used for each run. It should be noted, that our strategy is not intended for MPI programs, as it does not account for inter-node communication. However, these experiments give us a sense of the scalability of the proposed strategy. Fig. 7 shows the performance of multigrid, knapsack and advect3d as the number of cores is increased. We notice that although not quite linear, we do see significant performance gains as the number cores goes up. For multigrid and knapsack, performance plateaus at 16 cores whereas for advect3d we see very little improvement after 32. Simple hand calculations indicate that we should see performance gains for advect3d beyond what we see in these charts. We speculate that inter-node communication starts to interfere (and perhaps dominate) with performance when the implementation involves a large number of threads spread across many nodes.

![Overall Speedup Chart](chart.png)

5.3. Overall Speedup

Fig. 8 shows the overall best performance improvements achieved using our strategy on the three platforms. These numbers represent the best speedup obtained over the baseline parallel version when all optimizations in our scheme are applied. We see that among the three platforms, the best performance is obtained on 8Core. This can be attributed to both the larger number of cores and the smaller L2 cache size on this system. In terms of the three applications, knapsack yields the...
best performance on average, which is a direct result of the higher inter-thread data reuse present in this application.

6. Conclusions

This paper described a set of high-level language extensions that can be used in concert to improve the performance of pipeline-parallelize code on current multicore architectures. The language abstractions in conjunction with a source-to-source transformation can yield significant speed by by determining a suitable pipeline depth, exploiting inter-thread data locality and orchestrating thread affinity. Evaluation on three applications shows that the strategy is effective on three multicore platforms with different memory hierarchy and core configurations.

The experimental results reveal several interesting aspects of pipelined-parallel performance. We found that there is significant performance variation when the skew factor is changed and blocking in the outer dimension appears more critical than blocking in the inner dimensions. The experimental evaluation also points out some limitations of the proposed strategy. In particular, it was shown that inter-node communication can be an important factor influencing the performance of parallel pipeline applications.

References


