Adaptive Loop Tiling for a Multi-Cluster CMP

Jisheng Zhao, Matthew Horsnell, Mikel Luján, Ian Rogers,
Chris Kirkham and Ian Watson

{jishengz, horsnell, mikel.lujan, irogers, chris, watson}@cs.man.ac.uk
University of Manchester, UK

Abstract. Loop tiling is a fundamental optimization for improving data locality. Selecting the right tile size combined with the parallelization of loops can provide additional performance increases in the modern of Chip MultiProcessor (CMP) architectures. This paper presents a runtime optimization system which automatically parallelizes loops and searches empirically for the best tile sizes on a scalable multi-cluster CMP. The system is built on top of a virtual machine and targets the runtime parallelization and optimization of Java programs. Experimental results show that runtime parallelization and tile size searching are capable of improving performance for two BLAS kernels and one Lattice-Boltzmann simulation, despite overheads.

Keywords: Multi-Cluster CMP, Automatic Parallelization, Loop Tiling, Feedback-Directed Optimization.

1 Introduction

The tiling of loop iteration spaces is among the most popular and most extensvely studied automatic program optimization for improving data locality and cache performance [17, 6]. Selecting a suitable tile size is a critical step for improving performance. Some approaches have been proposed to calculate an optimal tile size for single processor architectures[7, 13].

In the context of CMPs [9, 12] and automatic parallelization, selecting the tile size not only affects cache performance but also the load balance among processors. For example, consider a 2-dimensional perfectly nested loop with a square $N \times N$ iteration space for which the optimal tile size is $\frac{N}{3} \times \frac{N}{3}$ for a given CMP using only one processor. When 4 processors are used in that CMP, load imbalance will occur using the same tile size; 9 tiles divided among 4 processors results in one processor receiving one extra tile.

Runtime optimization systems have the advantage of being able to observe the behavior of an executing application, whereas static compilers rely on predicting that behavior. Due to the limited amount of information available to a static compiler, optimizing parallel loop tiling for large CMP architectures can only become more and more complex. Performing runtime empirical searches, however, provide an alternative approach to improve the parallel execution of a program on different configurations of a CMP architecture.

Based on our previous research [19], at runtime it is feasible to automatically parallelize loops and also empirically search for adequate loop tiling sizes in CMP architectures with acceptable overheads. In this paper we concentrate on *multi-cluster CMPs* and whether adequate loop tiling sizes can be found at runtime for the automatically parallelized loops. As explained in further details in Section 2, processors are grouped together into clusters and the cache hierarchy is split into multiple levels which either connect the processors within a cluster or connect sets of clusters. The JAMAICA multi-cluster CMP [10] (see Figure 1) contains private L1 and multiple shared L2 caches. The L2 cache is unified containing both data and instructions, further complicating predictions as to how much space is available to data alone.For a multi-cluster CMP system which connects all the clusters by the L2 cache bus, the data locality in each L2 cache determines significantly the runtime performance. This paper investigates optimizations that search for multiple tile sizes to best utilize two levels of on-chip caching in a multi-cluster CMP, using runtime information to drive the search algorithm, in conjunction with an Online Tuning Framework (OTF) [19]. To exploit the cache hierarchy and the cluster structure two tile sizes need to be determined. The runtime tuning mechanism applies loop tiling recursively to target both clusters and cache levels.

The remainder of the paper is organized as follows. Section 2 gives a brief overview of the JAMAICA multi-cluster CMP architecture used in this paper. Section 3 describes the OTF and proposes the runtime tuning mechanism for optimizing the multiple tile sizes. Section 4 describes the experimental methodology. Section 5 presents and discusses the results from experimental evaluation. Section 6 presents related work, while a summary of the paper is presented in Section 7.

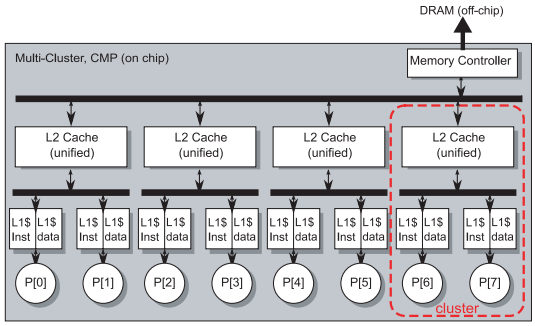

Fig. 1. JAMAICA: a multi-cluster CMP architecture.

2 JAMAICA Multi-Cluster CMP Architecture

To increase the ability of the JAMAICA architecture [2] to scale with the addition of more processing cores the single shared bus architecture is replaced by a scalable multi-level cache hierarchy [10] (shown in Figure 1).

The multi-level hierarchy is constructed by dividing the total number of cores into *clusters*. Each cluster contains a number of processing cores connected to a shared L2 cache. Each shared L2 cache is connected to a global on-chip memory network. This hierarchical approach can allow many more cores to be integrated onto a single chip, whilst maintaining shared memory, limiting the span of each interconnect to reduce the effects of cross-chip wire-delay, and with minimal design complexity.

Each intra-cluster network is independently arbitrated and accessed concurrently allowing the cores within each cluster to access the larger *cluster-shared* cache with less contention. The scalability comes at the expense of maintaining cache inclusion and the additional latency of sharing data between clusters. Such a hierarchy may be used to exploit an ever increasing transistor budget and as such is a feasible approach for future architectures.

3 Online Tuning Framework

The Online Tuning Framework (OTF) infrastructure, initially developed for CMP loop optimizations [19], performs automatic parallelization and enables runtime empirical search. It consists of three distinct elements: the Loop Parallelizing Compiler (LPC), the adaptive optimization component (see Section 3.1), and the runtime profiler (see Section 3.2).

The OTF is embedded within the adaptive optimization system (AOS) of the Jikes Research Virtual Machine (RVM) [4]. The Jikes RVM captures runtime information by instrumenting the running code at the method-level. Once the instrumentation indicates that a given method is hot (i.e. the number of times the method is executed is above a threshold), the AOS makes a decision whether to compile it using an optimizing compiler [5]. The OTF hijacks this decision, so that hot methods are also considered for parallelization by the LPC. The parallelized loop is reconstructed as a thread body which will be dispatched by a thread dispatcher method, as shown in Figure 2 (a). The procedure `loadConfiguration` loads the runtime configuration parameters (e.g. the loop tile size) from an AOS database. The `forkThreads` and `joinThreads` method calls create and synchronize those threads executing in parallel the loop body.

```
void threadDispatcher(.... ....) {
  loadConfiguration()
  forkThreads()
   ... ...
  joinThreads()
}
```

(a)

```
void threadDispatcherWithProfile(... ...) {
  long startCycle = getTimeBase();
  loadConfiguration();
  forkThreads()
   ... ...
  joinThreads()
  long executionCycle = getTimeBase() - startCycle;
  // Searching and Reconfiguration
  aosProcess(executionCycle, numIterations);
}
```

(b)

Fig. 2. Runtime profiling mechanism.

3.1 Adaptive Optimization Component

The Adaptive Optimization Component (AOC) applies one or more optimizations deemed to improve a loop identified by the LPC. Presently, the AOC

supports several adaptive optimizations for parallelizable code, although only adaptive tiling is described this paper.

Adaptive tiling is applied when a perfectly nested loop is identified by the LPC. In the current implementation, 2-dimensional loop traversals of the iteration space are divided into tiles which are then distributed among automatically generated parallel threads. We extend the basic empirical search algorithm [19] to vary the number of loop iterations inside each tile for the clusters and levels of the memory hierarchy. These parameters directly impact the balance between costs associated with thread management, the cache efficiency, and system load. As illustrated in Figure 3, the runtime reconfigurable parameters $L1Tile\_x$, $L1Tile\_y$, $L2Tile\_x$ and $L2Tile\_y$ are tuned using runtime empirical search. Note the parameters $cluster\_x$ and $cluster\_y$ in the two most outer loops. These divide the iteration space among processor clusters. However, these are not part of the search directly as they are determined indirectly by $L2Tile\_x$ and $L2Tile\_y$. The JAMAICA multi-cluster CMP architecture provides a cluster affinity mechanism to create parallel threads either on the local cluster or on remote clusters, which can be viewed as a potential extension to the pthread affinity in Linux/Unix. By splitting the loop iteration space, each cluster has its own thread creator that distributes the tiles to the processors.

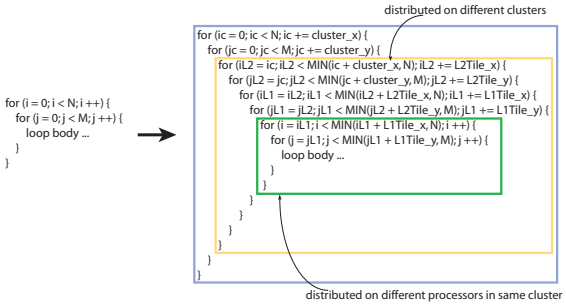

Fig. 3. Tiling transformation for runtime tuning.

Each tile has a corresponding divisor pair. Given a divisor pair (D_i, D_j), D_i is the divisor corresponding to the outer loop iterator and D_j corresponds to the inner loop iterator. Adaptive tiling uses a simple hill-climbing algorithm that starts from a divisor pair (D_{i0}, D_{j0}). The initial divisor pair is calculated such that $D_{i0} \times D_{j0} = P_n$, where P_n is the total number of processors. This partition, hereafter referred to as a naïve scheme, simply distributes the tiles evenly among the processing cores. Algorithm 2, included as an appendix, further describes how to calculate D_{i0} and D_{j0}.

The adaptive optimization component of the OTF, increases D_i and D_j iteratively to determine whether smaller tile sizes provide smaller execution times. When no performance improvement is observed, the OTF stops the search. Any divisor pair (D_i, D_j) calculated during iteration is composed such that $D_i \times D_j = k \times P_n$ where k is a positive integer value ($k > 0$) and P_n is the total number of processors. Algorithm 1, also included as an appendix, presents

the search algorithm used to determine the divisor pairs. The search space is a rectangular space which corresponds to the iteration space of a two-level nested loop. Each searching step shrinks the area of the tile by half or changes the shape of the tile.

For the specific multi-cluster CMP architecture considered, two tile sizes (for the L1 and L2 cache) are considered. The adaptive search begins by finding an optimal tile size for the L1 cache, which is a subset of the data within the L2 cache. When an optimal L1 tile size is determined, the OTF searches for a L2 cache tile size using the same searching algorithm but using different initial divisor pairs. Algorithm 3 describes the combined searching mechanism to optimize loop division for a multi-cluster CMP architecture. Recall the example loop shown in Figure 3, the search process for the L1 cache tile considers any rectangle which is contained within a rectangle with sides $cluster_x$ and $cluster_y$. The search space for the L2 cache tile is based on the L1 tile size. Given the optimal divisor pair for L1 tile, (D_x, D_y), the search process for the L2 tile is any rectangle with sides multiples of $\frac{cluster_x}{D_x}$ and $\frac{cluster_y}{D_y}$, respectively, and contained within the rectangle with sides $cluster_x$ and $cluster_y$.

3.2 Runtime Profiling and Overhead

To evaluate the performance of the applied optimizations is predicated upon access to runtime execution profile data. For adaptive tiling, this is achieved by using a profiling thread dispatcher, shown in Figure 2 (b). Two additional statements are inserted at the start and end of the parallelized loops. The first statement extracts from the architecture the cycle count[1] prior to the loops execution and the second statement extracts the cycle count after the loop has executed. The method `aosProcess` is responsible for reporting back to the AOS the total cycle count and the number of loop iterations per thread. The OTF is then able to calculate the execution time per iteration of each invocation of the loop and can make decisions about the comparative performance with other invocations of the same loop under different optimizations and configurations.

How representative are the measured execution times is a major factor for the success of the runtime empirical search, and there are two issues that affect it. The first one is that not all loops are of static length or duration. It is possible that both the number of iterations and the loop contents will vary from invocation to invocation. The second issue is that the execution timings are affected by system noise; for example, cold caches and other unrelated thread activity. To overcome these issues, the execution time for a given optimization on a parallelized loop is calculated, as an arithmetic mean of the cycles per iteration for three invocations of that loop. Loops that exhibit large profile deviations, defined as having a *coefficient of variation* (CV) [2] greater than a configurable threshold, for this work set at 0.1, are deemed unstable. When instability is

[1] Although this mechanism is machine specific; instructions exist in the main architectures: RDTSC (x86), mftb (PPC), TICK register (SPARC)

[2] Coefficient of variation is the ratio of the standard deviation to the arithmetic mean.

detected the profiling code is switched off and the current best optimization is used.

For each run of the parallelized loop, the profiling mechanism records and evaluates the timing data to progress or stop the search. The average overhead for each searching step is less than 300 cycles, thus the profiling overhead is nearly constant, although the accumulated overhead grows linearly with the number of searching steps. As tile size tuning is based on runtime modification of a set of parameters, there is no additional overhead for recompilation.

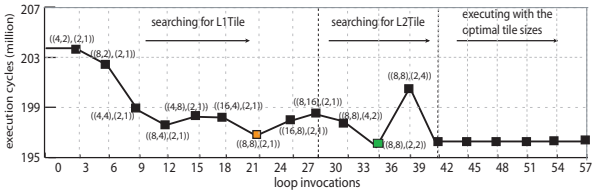

Fig. 4. Searching profile using DGEMM.

Figure 4 shows the OTF searching for an optimal divisor during the execution of the DGEMM benchmark. The problem iteration size is 256×256, the hardware is configured as a 2-cluster CMP architecture with each cluster containing 4 processing cores, from now on we refer to such a configuration using the notation: 2c/4p. The L1 cache size is 16KB, and 128KB for each L2 cache, again we will use the notation: 16KB/128KB. The searching algorithm starts from a naïve tile divisor: for the L1 tile $(4, 2)$, and for the L2 tile $(2, 1)$. By the $21th$ invocation of the parallelized loop a local optimal L1 tile size has been found, and a local optimal L2 tile size is found at the $18th$ invocation. Three invocations of the loop are used to assess timing stability. In this experiment the deviation did not exceed the threshold (0.1). The optimal L1 and L2 tile sizes are applied at the $36th$ invocation finishing the search phase. Note that by the very nature of the hill-climbing algorithms used, the adaptive searching finishes after finding locally-optimal solutions.

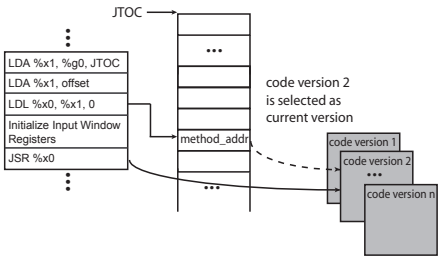

Fig. 5. Code version switching.

Once optimal divisors are found for loop tiling, the AOS switches off the runtime profiler and runs any subsequent executions of the loop using the best optimization found. The thread dispatcher, is switched to a version that does

not contain the timing instrumentation, so that future execution of the code runs without the cost of the profiling phase; see Figure 5. A runtime code patch mechanism is employed to redirect the execution path to the normal thread scheduler. A global data structure, `JTOC` [5], records references to both versions of the code. Each time the compiled code calls a routine, it is required to first load the method address from the `JTOC` (see the instructions `LDA` and `LDL`), and then jump to the loaded method address (see the instruction `JSR`), making such code switches possible.

4 Experimental Methodology

The experiments are performed on the multi-cluster CMP JAMAICA architecture [18], using the OTF as part of the adaptive optimization system of the Jikes RVM. The Jikes RVM has been ported to the JAMAICA architecture and runs directly on top of the hardware. The JAMAICA architecture is implemented within a highly configurable cycle-level processor and memory simulation platform. The simulation platform allows the evaluation of the OTF on a wide range of hardware configurations all using the same instruction set. The caches simulated are 4-way set associative.

```
for (int i = 0; i < mLength; i ++) {
   for (int j = 0; j < nLength; j ++) {
      double temp = 0.0;
      for (int k = 0; k < nLength; k ++) {
         temp += alpha * matrixA[i][k] * matrixB[k][j] + beta * matrixC[i][j];
      }
      matrixC[i][j] = temp;
   }
}
```

(a) DGEMM

```
for (int i = 0; i < length; i ++) {
   for (int j = 0; j < length; j ++) {
      double temp = 0.0;
      for (int k = i; k < length; k ++) {
         temp +=  matrixA[i][k] * matrixB[k][j];
      }
      matrixC[i][j] = temp;
   }
}
```

(b) DTRMM

Fig. 6. Level 3 BLAS kernels.

Two well known level 3 BLAS [3] kernels (DGEMM and DTRMM) and a 2D Java Lattice-Boltzamann (JLB) simulation (9 variables for each element) [1] are used in the performance evaluation. The kernels for DGEMM and DTRMM appear in Figure 6. Each kernel is executed to completion and validation on each simulated architectural configuration. The configurations assess the performance of the same optimizations in the presence of varying cache sizes, number of clusters and number of processors per cluster.

5 Results and Discussion

Different problem sizes (64×64, 128×128, 256×256 and 352×352 matrix) and different hardware configurations (clusters/processors: 2c/4p, 4c/2p and 4c/4p, and L1/L2 cache sizes: 8KB/128KB, 8KB/256KB, 16KB/128KB and 16KB/256KB) are used in the each experiment. For example, 4c/2p with 16KB/128KB refers to a multi-cluster CMP configured with 4 clusters each with 2 processors (total number of processors 8), and cache of sizes 16KB and 128KB for L1 and L2 cache, respectively.

The graph in Figure 7 presents the speedup attained using the optimal tile sizes compared with that attained using naïve tile sizes. The naïve tile size is defined as the square root of the number of processors. For example, a system with

16 processors has naïve L1 cache tile divisors $(4, 4)$. The divisors are restricted to integer values, thus in a system with 8 processors, the L1 cache tile could either be $(4, 2)$ or $(2, 4)$ (see Algorithm 2). The naïve scheme is in used by static optimizers as it achieves reasonable load balance and data locality. The results of this paper show that the performance can be further improved by performing a runtime empirical search.

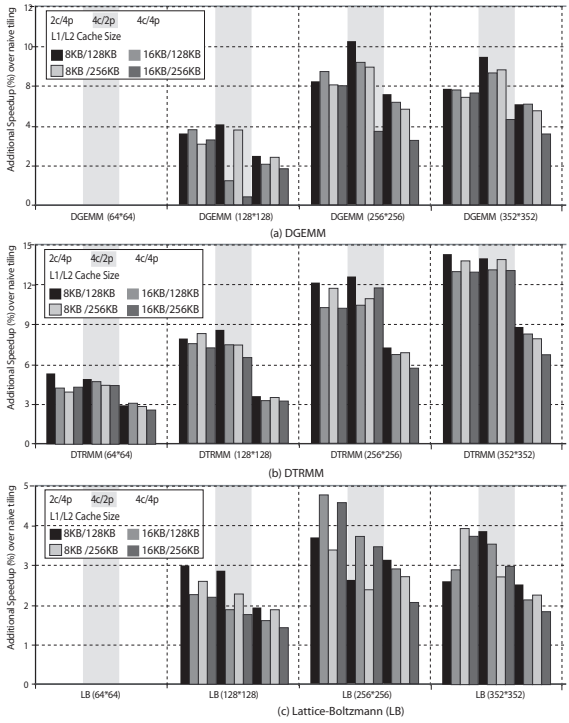

Fig. 7. The speedup compared with a naïve tiling scheme.

Figure 8 shows the resulting optimal divisors for all of the evaluated benchmarks. The naïve divisors for DGEMM are: 2c/4p and 4c/2p with L1 tile divisor $(4, 2)$ and L2 tile divisor $(2, 1)$, 4c/4p with L1 tile divisor $(4, 4)$ and L2 divisor $(2, 2)$. The speedup for DGEMM is shown in Figure 7(a). For small problem sizes (e.g. 64×64 matrix), there is no obvious benefit for those configuratins with larger L1 cache sizes when compared to the naïve scheme. For larger problem sizes, however, larger divisors produce performance increases. The optimal L2 tile sizes are related to the number of clusters. For example, the best L2 tile divisors for 64×64 matrix are $(2, 1)$ for the 2c/4p configuration and $(2, 2)$ for the 4c/2p and 4c/4p configurations. By increasing the L2 cache size, the L2 cache tile has less effect and its value is near the naïve configuration. This is why the 256KB L2 cache configurations used have lower speedup than the 128KB L2 cache configurations for the same problem size and L1 cache size.

The DTRMM nested loop is intrinsically load imbalanced, because the number of iterations in the inner most loop (k-loop) depends on the iteration of the i-loop, refer to Figure 6 (b). The optimal divisors are shown in Figure 8(b). For configurations 2c/4p and 4c/2p, most of the best divisors for the j-loop L1 tile sizes are 8, which is an even distribution of 8 parallel tasks to the 8 processing cores. Similarly, most of the best divisors for the j-loop L1 tile sizes are 16 for 4c/4p. By increasing the problem size, both the L1 and L2 divisors are increased to gain additional benefits from data locality. The speedups, shown in Figure 7(b), are more pronounced than for DGEMM, however, most of the benefit is attained through better load balancing.

	2c/4p				4c/2p				4c/4p			
Naïve Divisor	(4,2) (2,1)				(4,2) (2,2)				(4,4) (2,2)			
L1/L2 Cache Size	8KB, 128KB	16KB, 128KB	8KB, 256KB	16KB, 256KB	8KB, 128KB	16KB, 128KB	8KB, 256KB	16KB, 256KB	8KB, 128KB	16KB, 128KB	8KB, 256KB	16KB, 256KB
64*64	(4,2) (2,1)	(4,2) (2,1)	(4,2) (2,1)	(4,2) (2,1)	(4,2) (2,2)	(4,2) (2,2)	(4,2) (2,2)	(4,2) (2,2)	(4,4) (2,2)	(4,4) (2,2)	(4,4) (2,2)	(4,4) (2,2)
128*128	(4,4) (2,1)	(4,2) (2,1)	(4,4) (2,1)	(4,2) (2,1)	(4,4) (2,2)	(4,2) (2,2)	(4,4) (2,2)	(4,2) (2,2)	(4,4) (2,2)	(4,2) (2,2)	(4,4) (2,2)	(4,2) (2,2)
256*256	(8,8) (2,2)	(4,4) (2,2)	(8,4) (2,2)	(4,4) (2,2)	(8,4) (2,2)	(4,4) (2,2)	(8,4) (2,2)	(4,2) (2,2)	(8,4) (2,2)	(4,4) (2,2)	(8,4) (2,2)	(4,2) (2,2)
352*352	(8,16) (4,4)	(8,16) (4,4)	(8,16) (4,2)	(4,4) (4,2)	(8,16) (4,4)	(8,8) (4,2)	(8,16) (4,2)	(8,4) (4,2)	(8,16) (4,2)	(8,4) (4,2)	(8,16) (4,2)	(4,4) (4,2)

(a) DGEMM

	2c/4p				4c/2p				4c/4p			
Naïve Divisor	(4,2) (2,1)				(4,2) (2,2)				(4,4) (2,2)			
L1/L2 Cache Size	8KB, 128KB	16KB, 128KB	8KB, 256KB	16KB, 256KB	8KB, 128KB	16KB, 128KB	8KB, 256KB	16KB, 256KB	8KB, 128KB	16KB, 128KB	8KB, 256KB	16KB, 256KB
64*64	(1,8) (1,2)	(1,8) (1,2)	(1,8) (1,2)	(1,8) (1,2)	(1,8) (1,4)	(1,8) (1,4)	(1,8) (1,4)	(1,8) (1,4)	(1,16) (1,4)	(1,16) (1,4)	(1,16) (1,4)	(1,16) (1,4)
128*128	(4,8) (1,2)	(2,8) (1,2)	(2,8) (1,2)	(2,8) (1,2)	(2,8) (1,4)	(2,8) (1,4)	(2,8) (1,4)	(2,8) (1,4)	(1,16) (1,4)	(1,16) (1,4)	(1,16) (1,4)	(1,16) (1,4)
256*256	(4,8) (2,2)	(4,8) (2,2)	(4,8) (1,2)	(4,8) (1,2)	(4,8) (1,4)	(4,8) (1,4)	(4,8) (1,4)	(4,8) (1,4)	(2,16) (1,4)	(2,16) (1,4)	(2,16) (1,4)	(1,16) (1,4)
352*352	(8,8) (4,2)	(4,8) (4,2)	(8,8) (2,2)	(4,8) (2,2)	(8,8) (2,4)	(4,8) (2,4)	(8,8) (1,4)	(4,8) (1,4)	(2,16) (2,4)	(2,16) (2,4)	(2,16) (1,4)	(2,16) (1,4)

(b) DTRMM

	2c/4p				4c/2p				4c/4p			
Naïve Divisor	(4,2) (2,1)				(4,2) (2,2)				(4,4) (2,2)			
L1/L2 Cache Size	8KB, 128KB	16KB, 128KB	8KB, 256KB	16KB, 256KB	8KB, 128KB	16KB, 128KB	8KB, 256KB	16KB, 256KB	8KB, 128KB	16KB, 128KB	8KB, 256KB	16KB, 256KB
64*64	(4,4) (2,1)	(4,2) (2,1)	(4,2) (2,1)	(4,2) (2,1)	(4,2) (2,2)	(4,2) (2,2)	(4,2) (2,2)	(4,2) (2,2)	(4,4) (2,2)	(4,4) (2,2)	(4,4) (2,2)	(4,4) (2,2)
128*128	(4,8) (2,2)	(4,8) (2,2)	(4,8) (2,1)	(4,8) (2,1)	(4,8) (2,2)	(2,4) (2,2)	(4,8) (2,2)	(2,4) (2,2)	(4,8) (2,2)	(2,4) (2,2)	(4,8) (2,2)	(2,4) (2,2)
256*256	(4,8) (2,2)	(8,16) (4,2)	(8,8) (4,2)	(8,16) (4,2)	(8,16) (4,2)	(4,8) (4,2)	(8,8) (2,2)	(4,8) (2,2)	(8,8) (2,2)	(4,8) (2,2)	(8,8) (2,2)	(4,8) (2,2)
352*352	(8,8) (4,2)	(8,16) (4,4)	(8,8) (4,4)	(8,16) (4,2)	(8,16) (4,2)	(4,8) (4,2)	(8,16) (2,2)	(4,8) (2,2)	(8,8) (2,2)	(4,8) (2,2)	(8,8) (2,2)	(4,8) (2,2)

(c) Lattice-Boltzmann (LB)

Fig. 8. Optimal divisor pairs for different problem sizes and hardware configurations.

Finally the optimal divisors for JLB, which uses a stencil computation model, are compared to the naïve tile sizes, which are the same as for DGEMM. The speedups are shown in Figure 7(c). Compared with DGEMM, JLB has less cache cross-interference and as a consequence the optimization produces smaller speedups.

6 Related Work

Kisuki, O'Boyle and Knijnebury [11] investigated iterative compilation for loop tiling and loop unrolling. Their proposed compilation system achieved high speedups, outperforming static techniques. The system shows that high levels of optimization can be achieved in a limited number of iterations by applying a hill-climbing like searching algorithm.

The ATLAS project [16, 15] applies an automatic tuning mechanism to the BLAS (basic linear algebra software library). Given a BLAS operation, ATLAS uses empirical off-line searches relying on actual execution times to choose the best implementation on a specific architecture. ATLAS typically uses code

generators which generate multiple code versions, and has sophisticated search scripts to find the best choice. Despite being an off-line system, ATLAS guides the search using runtime profiling information.

Voss and Eigenmann [14] established an adaptive optimization framework named ADAPT which performs dynamic optimization on hot spots through empirical search. ADAPT uses dynamic recompilation to evaluate different optimizations and a domain-specific language to drive a search on the optimization space for a specific optimization. For example in loop unrolling, each level of unrolling is compiled, executed and timed, and the fastest version is kept and used for subsequent executions of the hot spot. The compiler used for recompilation is run on a separate parallel processor which reduces the recompilation overhead at runtime.

Fursin et al. [8] explored online empirical searches for scientific benchmarks. To reduce runtime code generation overheads, a set of optimized versions of code are created prior to the execution of a program. These versions are then evaluated at runtime with the best performing version chosen for subsequent execution. They employ predictive phase detection to identify the periods of stable repetitive behavior of a program and use these phases to improve the evaluation of alternative optimized versions.

In contrast with the above work, this paper combines a loop-level parallelizing compiler and an adaptive optimization framework, within a virtual machine, that targets a chip multi-cluster CMP architecture which has a multiple level cache hierarchy. The runtime optimization can leverage some strengths of iterative optimization to make JIT more suitable for CMP architectures.

7 Conclusion

Loop tiling is a fundamental optimization for improving data locality. As the use of CMPs increases, selecting the right tile size combined with the parallelization of loops within a virtual machine may be one way of increasing performance. This paper presents a runtime optimization mechanism which automatically parallelizes loops and tunes them for the best tile sizes on a scalable multi-cluster CMP, a feasible next generation multi/many core architecture.

By optimizing the tile sizes for both L1 and L2 caches, a memory intensive application can increase performance. The system is built on top of a virtual machine and targets the runtime parallelization and optimization of Java programs. Experimental results show performance speedups, up to 14.1% for two level 3 BLAS kernels (rectangular and triangular iteration spaces) and a 2D Lattice-Boltzmann simulation (stencil computation with 9 variables per grid point). The speedup is over a traditional parallelization and tiling scheme and also includes the initial overheads involved with profiling.

References

1. Lattice boltzmann method. http://www.latticeboltzmann.com/.
2. The Jamaica Project. http://www.cs.manchester.ac.uk/apt/projects/jamaica, May 2005.

3. E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.
4. Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, 2000.
5. M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings ACM 1999 Java Grande Conference*, pages 129–141, San Francisco, CA, United States, June 1999. ACM.
6. Steve Carr and Ken Kennedy. Compiler blockability of numerical algorithms. In *Supercomputing*, pages 114–124, 1992.
7. Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 279–290, New York, NY, USA, 1995. ACM Press.
8. Grigori Fursin, Albert Cohen, Michael O'Boyle, and Oliver Temam. Quick and practical run-time evaluation of multiple program optimizations. *Transactions on High-Performance Embedded Architectures and Compilers*, 1(1):13–31, 2006.
9. Lance Hammond, Benedict A. Hubbard, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. The Stanford Hydra CMP. *IEEE Micro*, pages 71–84, March–April 2000.
10. Matthew. J. Horsnell. *A chip multi-cluster architecture with locality aware task distribution.* PhD thesis, The University of Manchester, 2007.
11. T. Kisuki, P.M.W. Knijnenburg, and M.F.P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. *International Conference on Parallel Architectures and Compilation Techniques*, pages 237–246, 2000.
12. Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.
13. Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.
14. Michael Voss and Rudolf Eigenmann. High-level adaptive program optimization with ADAPT. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 93–102, 2001.
15. R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005.
16. R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.
17. Michael J. Wolfe. *High performance compilers for parallel computing.* Addison-Wesley, Redwood City, CA, 1996.
18. Greg Wright. *A single-chip multiprocessor architecture with hardware thread support.* PhD thesis, The University of Manchester, 2001.
19. Jisheng Zhao, Matthew Horsnell, Ian Rogers, Andrew Dinn, Chris C. Kirkham, and Ian Watson. Optimizing chip multiprocessor work distribution using dynamic compilation. In *Euro-Par*, pages 258–267, 2007.

8 Appendix - Adaptive Tiling Algorithms

Algorithm 1 tile_search_rect (search tile size for rectangle iteration space).

Input: Num, number of processors or clusters
Output: optimal divisor pair
Implementation:
step 1: $(D_{i0}, D_{j0}) \Leftarrow init\_tile\_rect(P_n)$;
Evaluate the runtime performance by initial tile size (D_{i0}, D_{j0}), get execution cycles E_0
step 2: $(D_i, D_j) \Leftarrow (D_{i0}, D_{j0})$
step 3: $(D_{il}, D_{jl}) \Leftarrow (D_i \times 2, D_j)$; $(D_{ir}, D_{jr}) \Leftarrow (D_i, D_j \times 2)$;
Evaluate the runtime performance by two tile sizes: (D_{il}, D_{jl}) and (D_{ir}, D_{jr}), get execution cycles E_l and E_r
if $E_0 \leq E_l$ and $E_0 \leq E_r$ **then**
  goto step 4
end if
if $E_r \leq E_l$ **then**
  $(D_i, D_j) \Leftarrow (D_{ir}, D_{jr})$
else
  $(D_i, D_j) \Leftarrow (D_{il}, D_{jl})$
end if
goto step 3
step 4:
if $(D_i, D_j) = (D_{i0}, D_{j0})$ **then**
  goto step 5
else
  return (D_i, D_j)
end if
step 5: $(D_i, D_j) \Leftarrow (D_{i0}, D_{j0})$; $i \Leftarrow 2$
step 6: $(D_{il}, D_{jl}) \Leftarrow (\lfloor \frac{D_i}{i} \rfloor, D_j \times i)$; $(D_{ir}, D_{jr}) \Leftarrow (D_i \times i, \lfloor \frac{D_j}{i} \rfloor)$;
Evaluate the runtime performance by two tile sizes: (D_{il}, D_{jl}) and (D_{ir}, D_{jr}), get execution cycles E_l and E_r
if $E_0 \leq E_l$ and $E_0 \leq E_r$ **then**
  return (D_i, D_j)
end if
if $E_r \leq E_l$ **then**
  $(D_i, D_j) \Leftarrow (D_{ir}, D_{jr})$; $E_0 \Leftarrow E_r$;
else
  $(D_i, D_j) \Leftarrow (D_{il}, D_{jl})$; $E_0 \Leftarrow E_l$;
end if
$i \Leftarrow i + 1$
goto step 6

Algorithm 2 init_tile_rect (initialize tile size for rectangle iteration space).

Input: Num, number of processors (or clusters).
Output: a initial divisor pair
Implementation:
step 1: $t \Leftarrow \lfloor sqrt(t) \rfloor$;
step 2:
while $(Num\%t)! = 0$ **do**
  $t \Leftarrow t - 1$
end while
step 3: return $(\frac{Num}{t}, t)$

Algorithm 3 Multiple levels search.

Input: P_n, number of processors; C_n number of clusters.
Output: optimal divisor pairs for L1 and L2 tile
Implementation:
step1: Search for L1 cache tile size
$L1Tile \Leftarrow tile\_search\_rect(P_n)$, $L1Tile$ is the optimal divisor pair for L1 tile
step2: Search for L2 cache tile size
$L2Tile \Leftarrow tile\_search\_rect(C_n)$, $L2Tile$ is the optimal divisor pair for L2 tile