Multi-objective Evolutionary Auto-tuning for Optimising Algorithm Speed and Cache Memory Usage

Darren M. Chitty
Department of Computer Science
University of Bristol, Merchant Venturers Bldg
Woodland Road, BRISTOL BS8 1UB
darrenchitty@googlemail.com

ABSTRACT

Modern CPUs are complex with hierarchical cache memory levels, vector instruction sets, instruction level parallelism and multiple processor cores. Hence, extracting the maximum performance for a given algorithm is a complex task and can require the optimisation of a number of parameters. This paper will demonstrate the use of an evolutionary approach to tune a matrix multiplication algorithm in terms of both execution speed and also cache memory usage. Moreover, it will be shown that these objectives conflict to some degree. Hence, a multi-objective evolutionary tuning approach is demonstrated that optimises for both of these objectives establishing a Pareto front of solutions.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Software—*Restructuring, reverse engineering, and reengineering*

General Terms

Algorithms, Experimentation, Performance

Keywords

Multi-objective Genetic Algorithms, auto-tuning

1. INTRODUCTION

This paper will demonstrate the use of an evolutionary auto-tuning approach to optimise an algorithm in terms of the separate objectives of execution speed and efficient cache memory usage. In doing so it will be shown that these two criteria are mutually exclusive. Thus, multi-objective evolutionary approaches will be investigated to optimise for both these criteria. To demonstrate multi-objective auto-tuning, the matrix multiplication algorithm will be used for its computational intensity and transparency to the reader.

Basic matrix multiplication has three nested loops, the first two loops iterating over each element of the resulting matrix with the inner most loop calculating the summation of the multiplication of each matching element of the respective row and column of the input matrices. However, this standard matrix multiplication algorithm does not exploit modern CPU architectures.

An improved design of a modern multi-core CPU matrix multiplication algorithm is shown in Algorithm 1. The first aspect of note is that the second matrix, B, is transposed to make better use of cache memory. Each time a location from main memory is requested, a copy is put in the faster cache memory along with neighbouring values which make up the *cache line*. By transposing the second matrix the access pattern becomes *row major* better exploiting cache memory.

Algorithm 1 Parallel Matrix Multiplication Kernel

```
Require: A, B^T, C, m, n, p, mDim, nDim, pDim
 1: for i=1 to m, i+=mDim do
 2:     for j=1 to n, j+=nDim do
 3:         for k=1 to p, k+=pDim do
 4:             for i2=i to i+mDim do
 5:                 for j2=j to j+nDim, j2+=2 do
 6:                     float Sum1=0, Sum2=0
 7:                     for k2=k to k+pDim, k2+=2 do
 8:                         Reg=A[i2][k2]
 9:                         Sum1+=Reg*B^T[j2][k2]
10:                         Sum2+=Reg*B^T[j2+1][k2]
11:                         Reg=A[i2][k2+1]
12:                         Sum1+=Reg*B^T[j2][k2+1]
13:                         Sum2+=Reg*B^T[j2+1][k2+1]
14:                     C[i2][j2]+=Sum1
15:                     C[i2][j2+1]+=Sum2
```

A second aspect to note is that there are now an extra three nested loops such that the matrix multiplication is now done in blocks [3]. Cache memory has a very limited size so when placing data into the cache memory, other data is pushed put. Generally, cache memory operates in a Least Recently Used (LRU) manner whereby the least recently used memory locations are pushed out of the cache. Since data in the input matrices is reused for multiple elements of the output matrix, by using blocks, this data is retained in the cache memory thus boosting performance. The three outermost blocks define the block size and the three inner most iterate over the elements of the defined block.

Another aspect that exploits the fact that memory access is much slower than the CPU is Instruction Level Parallelism (ILP). It can be considered that if an instruction is waiting for a memory request, another instruction could be executed

GECCO'13 Companion, July 6–10, 2013, Amsterdam, The Netherlands.

whilst the data is fetched from the memory. A method that exploits this is known as loop unrolling [2] which increases the number of instructions within a loop and reduces the number of iterations of the loop. Reuse of data in registers is another aspect that can speed up matrix multiplication as registers have the fastest access speed. Algorithm 1 demonstrates this principle with a register used to store an element of matrix A which is then reused. Finally, the Advanced Vector Extensions (AVX) instruction set is used which enables a limited level of data parallelism with multiple data values contained in a 256 bit register and a single operation performed on the register executes on these multiple data values simultaneously. The kernel shown in Algorithm 1 can be executed in parallel with separate blocks of the resulting matrix calculated by differing threads.

Subsequently, there are six parameters which need to be optimised. The three dimensions of the cache block size, the number of parallel threads to be used, the degree of loop unrolling and the amount of register reuse. An additional two parameters are used to specify the cache block size used for the matrix transpose. An evolutionary algorithm is used to tune these parameters to minimise the execution speed using an Intel i7 processor on multiple matrices. Table 1 shows the speed achieved by the optimised algorithm on four matrix sizes along with the cache misses.

The experiment is now repeated but instead of minimising execution speed, the goal is to minimise the cache memory misses over all three levels of cache memory. These results are also shown in Table 1. From these results it can be observed that the speed achieved is significantly slower but the cache misses are also significantly lower. Thus, it can be considered that the two objectives of minimising cache memory misses and minimising execution speed are conflicting.

Table 1: The timings and cache misses for matrix multiplication optimised separately for minimising speed and cache misses.

Objective	Matrix	Approx.	Cache Misses (millions)		
	Size	**GFlops**	L1	L2	L3
Optimal Speed	1000	92.815	0.787	0.435	0.180
	2000	96.066	7.462	3.321	1.455
	3000	96.900	22.661	10.769	4.249
	4000	97.508	32.023	25.035	9.612
Optimal Cache	1000	21.359	0.572	0.263	0.208
	2000	23.568	3.326	1.534	0.973
	3000	23.546	7.952	4.148	2.527
	4000	23.330	27.916	15.592	9.270

2. AN EVOLUTIONARY MULTI-OBJECTIVE APPROACH TO AUTO-TUNING

With two mutually exclusive objectives, it is not possible to minimise one objective without increasing the other. A given solution x can be said to dominate solution y if and only if $f_i(x) <= f_i(y)$ for $i = 1, ..., K$ where K is the number of objectives and also $f_j(x) < f_j(y)$ for at least one objective j. A set of non-dominated solutions is known as a Pareto front. The evolutionary algorithm used to find the Pareto optimal set of algorithm solutions is the NSGA-II algorithm [1]. The NSGA-II algorithm is a Pareto ranking technique and operates by identifying the set of non-dominated fronts within a population of individuals. A Pareto optimal front is identified and the solutions removed from the population. The process repeats until no individuals are left. Solutions

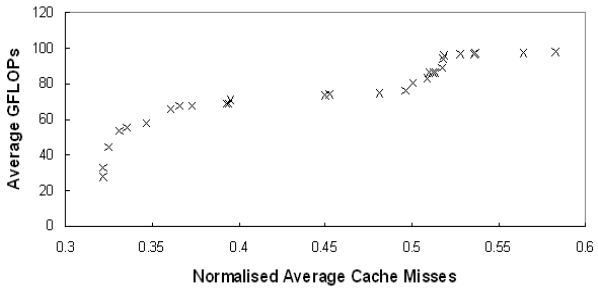

Figure 1: An example Pareto front of evolved matrix multiplication algorithms

are ranked in terms of which front they belong to. A second fitness measure is used if two solutions belong to the same front with the Euclidian distance calculated for each solution to all the other solutions within the front. The solution with the highest distance is considered the fitter as this promotes diversity with solutions on less crowded aspects of the front being rewarded.

Table 2: The timings and cache misses for both optimal objectives from the Pareto front

Objective	Matrix	Approx.	Cache Misses (millions)		
	Size	**GFlops**	L1	L2	L3
Optimal Speed	1000	89.915	0.579	0.418	0.166
	2000	94.583	5.359	3.340	1.450
	3000	95.691	16.310	10.946	4.237
	4000	97.887	31.935	25.002	9.658
Optimal Cache	1000	28.115	0.602	0.204	0.148
	2000	26.285	3.986	1.704	0.979
	3000	26.192	10.463	4.810	2.880
	4000	29.475	21.054	10.419	6.375

Using the NSGA-II algorithm, an optimal Pareto front of solutions is found which is shown in Figure 1. From this it can observed that the two objectives of minimising execution time and cache misses are indeed mutually exclusive. The two solutions from either end of the Pareto front are tested on the matrix multiplication problems with the results shown in Table 2. For the solution which had optimal speed, the GFLOPs are only slightly worse than those for the optimised single objective of execution speed shown in Table 1. However, the cache misses are reduced over all cache levels. In terms of the optimal cache efficient solution, when comparing with the optimised single objective of reducing cache misses in Table 1, the cache misses are broadly similar and in the case of the largest matrix size better performance is observed. Moreover, the execution speed is approximately 20% better than when optimising for cache memory efficiency alone.

3. REFERENCES

[1] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *Trans. Evol. Comp*, 6(2):182–197, Apr. 2002.

[2] J. Dongarra and A. R. Hinds. Unrolling loops in FORTRAN. *Softw., Pract. Exper.*, 9(3):219–226, 1979.

[3] G. Golub and C. Loan. *Matrix computations*. Johns Hopkins series in the mathematical sciences. Johns Hopkins University Press, 1989.