Comparison of Cilk, Kaapi and CUDA for the Jacobi Method

Luís Felipe Millani, Nicolas Maillard
Instituto de Informática - Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
{lfgmillani, nicolas}@inf.ufrgs.br

Abstract

While multi-core architectures allow for high performance gains, the use of a specialized framework is often required to maximize the use of the available hardware. This paper compares the performance of three different frameworks, Cilk, Kaapi and CUDA, in implementing the Jacobi method.

1. Introduction

With multi-core processors widely available and clock speeds approaching physical limits, computationally intensive applications must make use of parallelism. However, the development of parallel applications often requires the use of a different programming model. Frameworks like Cilk and Kaapi aim to ease the development of parallel applications by managing the division of the tasks over the different processing units [3] [4].

Both Cilk and Kaapi use a work-stealing policy for the mapping of tasks to processors. Unlike work-sharing schedulers, which do this mapping when the tasks are created, a work-stealing scheduler will have the idle processors actively attempt to ”steal” tasks from other processors [1].

Problems which can be divided into a very large number of similar tasks often perform better in a higly-parallel architecture such as a GPU [6]. In those cases a programming model such as CUDA is more appropriate.

In this paper we compare the performance of Jacobi method implemented using Cilk, Kaapi and CUDA.

1.1. Jacobi Method

The Jacobi method solves a matrix equation in the form $Ax = B$, with A and B known and x unknown, given that the matrix A has no zeros in its diagonal [2]. Furthermore, the method always converges for any initial x if at least one of the following conditions hold

$$max_k \sum_{i=1,i\neq k}^{n} \left|\frac{a_{ik}}{a_{ii}}\right| < 1$$

$$max_i \sum_{k=1,k\neq i}^{n} \left|\frac{a_{ik}}{a_{ii}}\right| < 1$$

The method approximates the solution by with the following iterating rule, where μ is the iteration index

$$x_i^{\mu+1} = \frac{b_i}{a_{ii}} - \sum_{k=1,k\neq i}^{n} \frac{a_{ik}}{a_{ii}} x_k^{\mu}$$

1.2. Algorithm

The algorithm used to implement the Jacobi method is as follows

```
1:  $D^{-1} \leftarrow$ inverse of the diagonal of $A$
2:  $d^{-1} \leftarrow$ vector of the diagonal components of $A$
3:  $L \leftarrow$ lower triangular of $A$
4:  $U \leftarrow$ upper triangular of $A$
5:  $currentX = \{0 \dots 0\}$
6:  $\mu \leftarrow 0$
7:  repeat
8:      $nextX \leftarrow (L + U) \times currentX$
9:      $nextX \leftarrow nextX \times -d^{-1}$
10:     $nextX \leftarrow nextX + (-D^{-1} \times b)$
11:     $nextX, currentX \leftarrow currenX, nextX$
12:     $\mu \leftarrow \mu + 1$
13: until $\mu = numIter$
```

2. Implementation

Despite the fact that Kaapi has support for heterogeneous architectures, with use of CPUs and GPUs [5], we didn’t make use of this feature. As such, our Cilk and Kaapi implementations were restricted to the CPU, and the CUDA implementation to the GPU.

Function	% Execution time
Matrix product	99.44%
Read input	0.43%
Vector addition	0.05%
Hadamard product	0.05%
Other functions	0.03%

Table 1. Profiling results for the sequential version with a 1024×1024 matrix

2.1. Profiling

Before parallelizing the application we profiled the sequential version to find where most of the execution time was spent. This was done by executing the sequential version with a 1024×1024 input matrix under the profiling tool Valgrind. As can be seen in table 1, the profiling results show that more than 99% of the time spent by the application is spent doing matrix vector products. As parallelizing the other portions of the application would give meager gains at best, we opted to only parallelize the code pertaining to calculating the matrix vector product.

3. Experiments

3.1. Methodology

The experiments where executed on a computer with an Intel Core i7 930 CPU (2.80 GHz), with four physical cores, a NVIDIA GeForce GTX 480 GPU and 12GiB of memory. Each experiment was repeated 20 times and the median result was used. In all implementations we used single-precision floats (4 bytes).

3.2. Parameters

All experiments were executed for 10000 iterations. Each subtask of the matrix product computed 128 elements of the resulting vector. That size was chosen after ad-hoc tests with values ranging from 8 to 256.

3.3. Results

As can be seen in the figures 1, 2, 3, 4 and 5, Cilk and Kaapi had equivalent results for matrices of sizes 2048×2048 and larger, with Kaapi outperforming Cilk on smaller problems.

On table 2 Kaapi outperforms Cilk and CUDA on small matrices, and CUDA outperforms Cilk and Kaapi on larger matrices. We attribute CUDA's low performance on small matrices to the overhead caused by the CUDA kernel calls. With larger matrices the overhead is relatively smaller, allowing for significant performance gains.

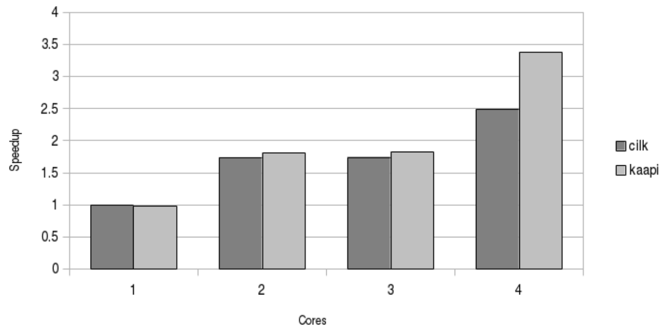

Figure 1. Results for Cilk and Kaapi with a 512×512 matrix

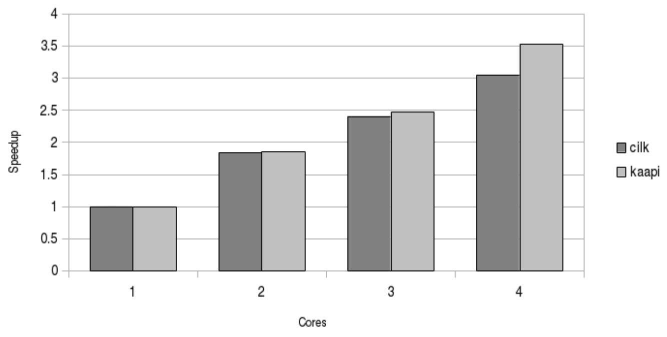

Figure 2. Results for Cilk and Kaapi with a 1024×1024 matrix

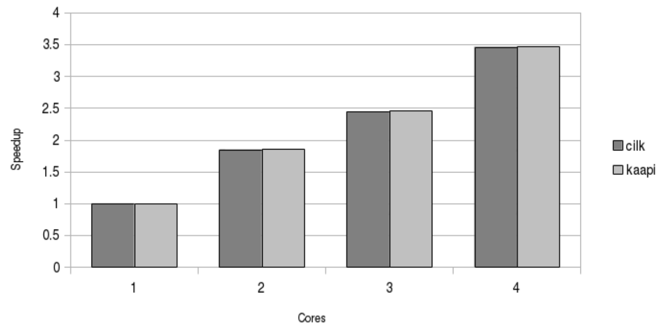

Figure 3. Results for Cilk and Kaapi with a 2048×2048 matrix

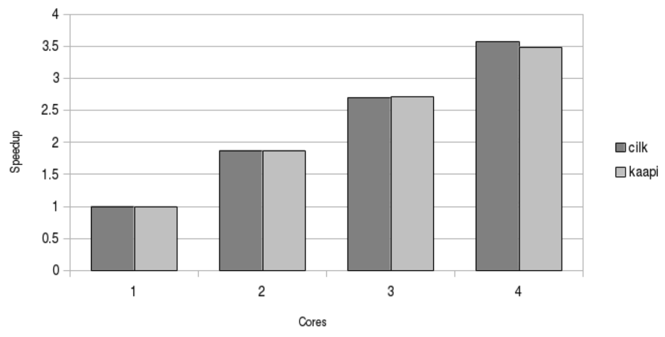

Figure 4. Results for Cilk and Kaapi with a 4096×4096 matrix

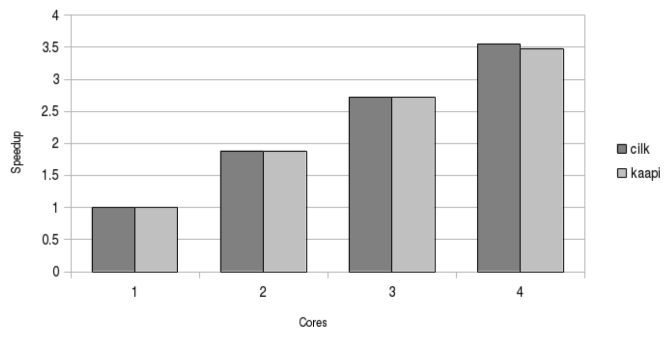

Figure 5. Results for Cilk and Kaapi with a 8192×8192 matrix

Matrix size	Cilk	Kaapi	CUDA
512	2.48	3.37	0.94
1024	3.04	3.52	1.81
2048	3.46	3.46	23.26
4096	3.56	3.48	73.03
8192	3.55	3.48	155.13

Table 2. Median speedup results, with Cilk and Kaapi limited to 4 processes

4. Conclusion

For tasks of similar sizes, Cilk and Kaapi show similar performance gains. Kaapi has a lower overhead when few tasks are created, whereas for a large number of tasks the speedup is very similar, with Cilk showing slightly better results.

Considering only performance, with no regards to easiness of development, Cilk and Kaapi are equally good choices for large problems. That is, however, assuming the tasks are of identical sizes and that the more advanced features of Kaapi are not used.

As expected, CUDA's GPU code outperforms the CPU implementations in matrix multiplication.

References

[1] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[2] I. N. Bronshtein, K. A. Semendyayev, G. Musiol, and H. Mühlig. *Handbook of mathematics*, page 895. Springer, 2007.

[3] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *ACM Sigplan Notices*, 33(5):212–223, 1998.

[4] T. Gautier, X. Besseron, and L. Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 15–23. ACM, 2007.

[5] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard. Multi-gpu and multi-cpu parallelization for interactive physics simulations. In *Euro-Par 2010-Parallel Processing*, pages 235–246. Springer, 2010.

[6] C. Nvidia. Nvidia cuda programming guide, 2011.