Lab Project Report[*]

Heat Dissipation on a Cylindrical Surface

Assignment 1

Christopher Esterhuyse, Mihai-Andrei Spadaru

1 Introduction

The heat dissipation on the surface of a cylinder is continuous by nature in both the spacial and temporal domains. However, the process can be sufficiently modeled when both domains are discretized if sufficient resolution is used.

Here we simulate such models using an implementation in C. Although it is only a modestly complex computational problem, it is complex enough to facilitate many interesting avenues of optimization. In this lab report, we explore a number of such optimizations, and observe the performance of the system in response.

2 Implementation Design Decisions

Even a sequential implementation affords some design decisions. Here we elaborate on some choices, their implications on preserving correctness, and their impact on performance.

2.1 Padded Matrices

As both dimensions of a 2D matrix increase, the total number of cells increases *quadratically*, while the number of cells along the boundary increases only *linearly*. As such, for any sizable matrix, an arbitrary cell has a small chance of being on the boundary.

Boundary cells require a unique variation of the local heat dissipation calculation to account for missing neighbor cells on one or more sides. Our implementation avoids a conditional branch by iterating over a matrix after it been *padded* on both vertical and horizontal sides with one extra layer of cells. All other cells in the matrix are henceforth referred to as *interior* cells.

Cells on vertical boundaries (corresponding with the top and bottom of the cylinder) retain constant values after being initialized. Cells on horizontal boundaries (corresponding with the continuously-curved surface of the cylinder) contain values that *mirror* the values of the interior cells on the opposite edge. Maintaining these mirrored cells incurs only an additional cost linear in the height of the cylinder per iteration step.

2.2 Cache Coherence

The workhorse of the program is a doubly-nested loop that traverses all interior cells once per iteration. To leverage memory locality more effectively, the coarser outer loop iterates over rows, while the inner loop iterates over adjacent (horizontal) cells, which are more likely to share cache lines. In a small experiment, it was observed that iterating over cells vertically first resulted in an order of magnitude slowdown (FLOPs reduced from 3.48×10^6 to 3.50×10^5).

The user's input parameters (for example, 'niter', bounding the maximal number of simulation iterations) are passed by reference into the main computation function. As the parameters never change, their values are copied into local variables to avoid needlessly-repeated pointer traversals.

2.3 Inner Loop Optimizations

Within the innermost loop, the heat calculation for a given cell is extremely performance critical. To

[*]This report was submitted for the lab assignment of the 2018 edition of the Universiteit van Amsterdam Programming Multi-Core and Many-Core Systems master course.

minimize cost, some needed values are defined as compile-time constants (such as $sqrt(2) + 1$), and a division operation is substituted with a faster multiplication of the reciprocal. The utility of this approach was demonstrated by an anecdotal slowdown[1] when multiplications are naively distributed over neighbor cells *before* neighbor value aggregation.

Additionally, it was verified (by a small, definitive experiment[2]) that this function was being correctly *inlined*.

2.4 Custom Abs Function

An attempt was made to use a custom double absolute value function in place of the C standard library function 'fabs'[3]. The custom function was intended to use bit-masking to avoid a conditional branch. However, further investigation suggested that this approach would lead to a brittle implementation, with correctness dependent on innocuous compiler behavior. Instead, the custom fabs function was identical to the standard library version, but with an added inline tag.

3 Sequential Benchmarks

The performance of out sequential implementation is measured as the number of floating point operations per second (FLOPs). Precisely which properties of the input matrices influence the performance of our program was not known *a priori*. The dimensions of the input serve as suitable starting points for experimentation, as length and breadth can be measured easily and can be intuitively understood. However, the two dimensions alter the iteration pattern of the matrix in different ways, and together alter other interesting properties of the problem as a whole (such as the ratio of boundary cells to interior cells). In this section, we attempt to understand the relationship between these two dimensions of the input matrix and their relationship to the achieved FLOPs.

[1] A speedup of 29% was achieved with the faster of two options on an input problem with dimensions 100×150

[2] The total time of a given run (repeated five times) was perceived to change from 0.241 to 0.400 sec without inlining.

[3] Available at url: `http://web.mit.edu/freebsd/head/lib/libc/mips/gen/fabs.c`

(a) Initial temperature (b) Conductivity

Figure 1: Initial temperature and conductivity inputs for all experiments in Sections 3 to 5.

To preserve the other properties of the problem across all image sizes, all experiments were run using scaled[4] versions of the same initial temperature and conductivity images, seen in Fig 1. These images were selected as they are complex enough to avoid the simulation reaching equilibrium too quickly, as well as exemplifying problems with both smooth gradients and sharp edges.

3.1 Square-Surface Matrices

First, we measure the effect of varying only the problem size ie. scaling both dimensions in proportion, thus using only square input matrices. The results can be seen in Fig 2.

From the plot, it can be seen that although the scales of the input images span several orders of magnitude, the differences in performance do not. A problem instance with 20000% more matrix cells achieved only 10.4% fewer FLOPs. Nevertheless, larger problem instances were associated with a steady decrease, which may be attributed to smaller fractions of the matrix fitting into the machine's cache lines at a time.

3.2 Rectangular-Surface Matrices

Next, we observe what effect varying the *ratio* between the two dimensions of the input cylinder has on the program's performance. This experiment was performed twice, each time fixing a different dimension at the arbitrarily-chosen value of 100. The results can be see in Figs 3 and 4.

From these results it can be seen that the program was most performant when tasked with narrower, taller cylinders. This is likely a result of the finite size of the various cache lines available to the

[4] All images were scaled using linearly-interpolated versions of a starting image with dimensions 500×500.

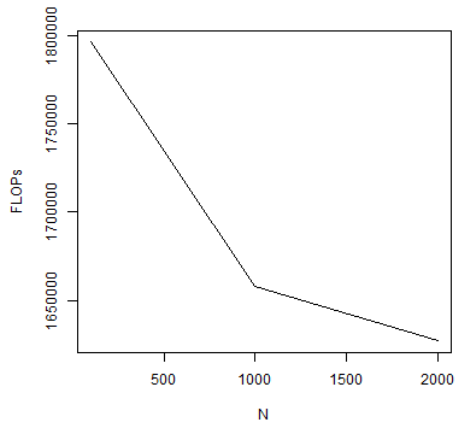


Figure 2: Average FLOPs achieved with inputs of dimensions $N \times N$ for N in $\{100, 1000, 2000\}$.

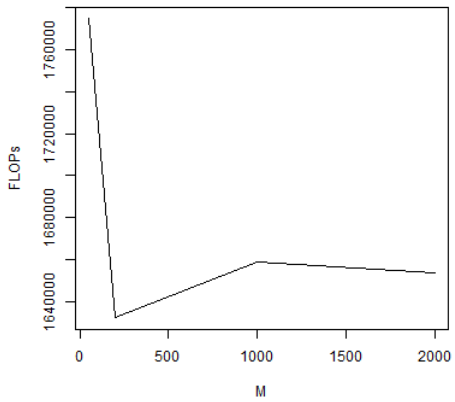


Figure 3: Average FLOPs achieved with inputs of a fixed height of 100 and breadth in $\{50, 200, 1000, 2000\}$.

hardware. When matrix rows are shorter, it becomes more likely for cells of the *next* row down to be within the same cache line as the current row. However, the benefits of this are marginal, as can be seen from the small difference in FLOPs across all experiments.

It is also worth noting that these results suggest that the performance costs of the mirroring horizontal boundary cells described in Section 2.1 (whose cost scales linearly with cylinder *height*) were overshadowed by other contributors to the overall performance.

4 Compiler Optimizations

The performance of the program is a function of the various optimizations used by the compiler when generating the binary. In this section, we explore the performance of the system in solving a single problem repeatedly, but when compiled with two well-known compilers: The GNU Compiler Collection (gcc) and the Intel C Compiler (icc).

Both gcc and icc expose various *optimization levels* as options to the user at compile time. 'Under the hood' such optimization levels are translated to sets of optimization passes that reflect a certain trade-off between compiler speed on the one hand, and execution speed and/or binary size on the other

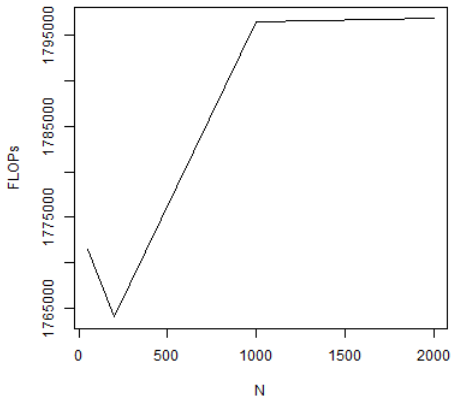


Figure 4: Average FLOPs achieved with inputs of a fixed breadth of 100 and height in $\{50, 200, 1000, 2000\}$.

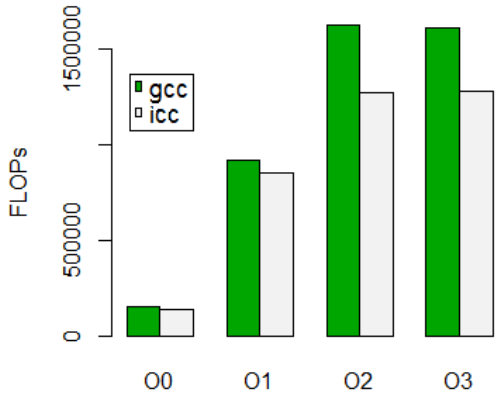


Figure 5: FLOPs achieved on the same problem instance with the binary compiled using all combinations of compilers in $\{gcc, icc\}$ and optimization levels 0 to 3.

hand[5]. For both gcc and icc, these levels range from 0 (fastest compilation) to 3 (fastest execution).

All optimization-level-to-compiler combinations were tested using the same input problem displayed in Fig 1, with a problem instance with dimensions 2000×2000. The resulting performance measurements are shown in Fig 5.

Using either compiler, there is little appreciable difference between levels 2 and 3. Level 2 is known to apply optimization passes that are almost certainly worthwhile ie. considered 'safe bets'. The third level introduces some extra optimizations on top (such as loop unrolling and automatic function inlining). From some experiments, we were able to confirm that all the desired functions were already being inlined by level 2. This is supported further by the diminishing returns after level 2 in Fig 5.

In this case, gcc performed consistently better than icc at generating *fast* code. Both icc and gcc allow the user to delve further into using specific optimization flags. Icc in particular allows enabling of optimization passes that are processor-specific. In-depth knowledge of these options and their properties would allow a user to potentially allow the icc-compiled binary to outperform that of the gcc-compiled binary.

[5]Of course it is not always that simple. There are indeed cases where specific optimization passes are undesirable, however this is usually not the case.

5 Vectorization

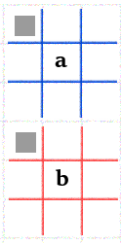
Fig 6a illustrates how temperatures values for neighboring matrix cells were thus-far computed independently; each neighbor is traversed sequentially each time.

Vectorization is a well-known method of fitting the same logical computations into fewer CPU clock cycles. Opportunities for its use arise in the event of any form of *data-parallelism*. From experiments in previous sections, even using optimization level 3, neither gcc nor icc were able to identify opportunities to leverage any meaningful vectorization for our program.

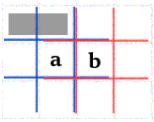
Manual vectorization by use of *compiler intrinsics* was employed to compute temperature values of neighboring cells in parallel. As seen in Fig 6b, with vectorization, the usual sequential approach is mimicked in a two-fold parallel computation of neighboring cells by traversing the same *relative* neighbors for two adjacent cells in lockstep. Once loaded, the vector registers can perform the summation and multiplication operations in tandem. In the event the breadth of the input matrix is not divisible by two, the remainder cells have their values computed in the sequential manner.

The vectorized program was pitted against the sequential program for the same task from Section 3.1. Results are shown in Fig 7. Unfortunately, the vectorized implementation was outperformed by the sequential implementation. It may be the case that, given wider vectors, and thus the ability to further unroll the inner for-loop, it might provide the edge needed to surpass the sequential implementation.

The reason for this slowdown is unclear. The memory-access patterns of the vectorized code is not sufficiently different enough to suggest significant slowdown. It was also confirmed that both versions of the program were benefiting from inlined functions as intended. The most likely perpetrator is the differences in *structure* between versions of the program, resulting in the sequential program enjoying additional compiler optimizations that are not also granted to the vectorized program.



(a) Without vectorization



(b) With vectorization

Figure 6: Effect of computing the temperature values of two adjacent cells. With vectorization, the respective neighbor cells values are computed in parallel lockstep.

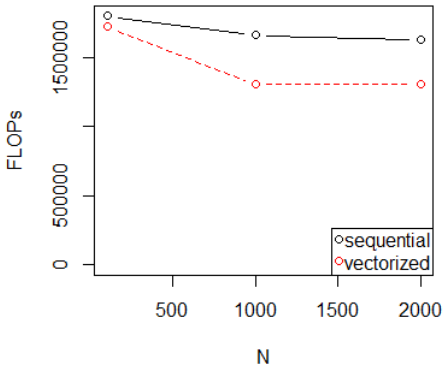


Figure 7: Average FLOPs achieved with inputs of dimensions $N \times N$ for N in $\{100, 1000, 2000\}$. with the performance of the vectorized program compared to that of the sequential program.

6 Conclusion

Optimization for speedup of even such a simple program provide plenty of challenges. From our experience, best results come from a two-way handshake between programmer and compiler. By writing code in a manner that *facilitates* compiler optimization, the work of generations of compiler programmers can shine through in an order of magnitude speedup that seems to come from thin air.

Vectorization seems to offer the potential for speedup that is greater still, but comes with the caveat of complicating the code and relying on the presence of specific extensions to the instruction set architecture. In cases where this requirement is satisfied, vectorization can provide a boost to speedup on top of that which came before.