CS 530: High-Performance Computing

Optimizing Matrix Multiplication

Nathan Chapman

Central Washington University

June 2, 2024

Contents

1 Introduction

There are many ways to decrease the execution time of a program. The biggest source of such a decrease is to execute the program in parallel. Though, not all program can benefit from being run in parallel. In such sequential cases, decreases in execution time can be gained from other sources, such as optimization of sequential implementations. If these optimizations are to be implemented, there are two ways that can be done: automatically by the compiler, or manually structuring the source code in such a way so-as-to not fall into the traps of inefficiency. There are many compilers that can be used to process source code either written in C/C++ or Fortran, possibly the most popular being the GNU compiler collection (gcc). gcc comes with a suite of different levels of automatic optimzations that be easily passed without further consideration. If the source code is to be optimized manually, there are several concepts to be considered when implementing algorithms. These include, but are not limited to, ideas such as how many values can be held in a single CPU-register at a time (which can handled with loop-fission or -fusion), doing the same operation over and over again inside a loop (which can be handled with methods such as loop peeling), the cost of calculating versus loading a value from memory. These techniques, whether using automatic compiler optimizations or writing manually optimized code, can provide programs that execute in substantionally less time than their "naive" un-optimized counterparts.

2 Methods

2.1 Matrix Multiplication

The mathematical definition of matrix multiplication takes the form

$$(AB)_{i,j} = \sum_{k=1}^{L} A_{ik} B_{kj} \tag{1}$$

where A, B are $N \times L$ and $L \times M$ matrices, respectively, and $(AB)_{i,j}$ is the element of the i-th row and j-th column of the product. Because this definition is for N rows, M columns, and L elements in the sum, the total number of either additions or multiplications is $N \times M \times L$. For the multiplication of an $N \times N$ square matrix with itself (a "matrix power"), this leads to N^3 operations for a single multiplication. If this is done multiple times p to create an arbitrary matrix power A^p of an $N \times N$ matrix A, the number of total operations needed is pN^3. Using asymptotic notation, this means that matrix powers are $\mathcal{O}(pN^3)$. The algorithm for matrix multiplication and matrix power are shown in algorithms 1 and 2, respectively.

Algorithm 1: Matrix Multiplication

Input: $N \times L$ Matrix A, $L \times M$ Matrix B
Output: $N \times M$ Matrix C
for $i \leq N$ **do**
  for $j \leq M$ **do**
    for $k \leq L$ **do**
      C[i,j] ← C[i,j] + A[i,k] * B[k,j]
    end
  end
end

Algorithm 2: Matrix Power

Input: $N \times N$ Matrix A, Integer $p > 0$
Output: $N \times N$ Matrix P
P ← A
for $i \leq p - 1$ **do**
  P ← multiply P and A using matrix multiplication
end

In order to test the efficiencies of the Fortran, such matrices need to be initialized. For this investigation, these matrices were initialized using random numbers. The default psuedo-random number generator in Fortran is xoshiro256**. All elements of the initial matrices are between 0 and 1; negative numbers could also be used to mitigate overflow in their data-types as the elements of the matrix-product grow astronomically large.

2.2 Automatic Optimization

There are many ways to automatically optimize source code of a program. When using a compiled language such as C/C++ or Fortran, optimizations can be done automatically by the compiler. In the case of the GNU Compiler Collection (gcc), there are several easily-passed, command-line flags. While there are many options for fine tuning, there are just a few notable options that enable entire collections of these options for different levels of optimization. These options are given by the `-On` flag, where `n` is 0,1,2,3, or `fast`. Each of these, in increasing order, provide more optimizations and let the compiler further handle and manipulate the source code once pre-processing is started. Below is a list of summaries of these optimization flags (as used in gcc and gfortran) with a few of the more-easily-understandable fine-tuning options that are applied.

- `-O0`
  - No optimization of the given source code is done.
  - The default option i.e. the option used when no other optimization flag is given
  - This level of compiler optimization was used in calculating previous results.
- `-O1`
  - Provides slight optimization of the source code
  - Reduces the size of the generated assembly code
  - Reduces the length of time the program takes to execute
  - Only provides changes that will not increase compile time by a large amount
  - `-finline-functions-called-once`: If a region of code only calls a specific function a single time, the compiler replaces the function call with the body of the function (known as *inlining*). This reduces execution time because there are fewer queries to memory and fewer allocations on the call-stack.
- `-O2`
  - Performs nearly all supported optimizations that do not involve a space-speed tradeoff.
  - Reduces the length of time the program takes to execute
  - Increases the amount of time the program will take to compile
  - Includes all flags of `-O1`
  - `-finline-functions` considers all functions for inlining, automatically determining which are worth inlining.
  - `-finline-small-functions` inlines functions when including the explicit body would result in fewer operations than including a call to the function.
  - `-findirect-inlining` inlines calls generated by `finline-functions` or `-finline-small-functions`.
  - `-fpartial-inlining` inlines *parts* of functions when generated by `finline-functions` or `-finline-small-functions`

- `-O3`
  - Performs many optimizations at the cost of a considerably greater compilation time
  - Includes all flags of `-O2`
  - `-floop-interchange` interchanges nested loops. This flag can improve cache performance on loop nest and allow further loop optimizations, like vectorization, to take place. For example

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    for (int k = 0; k < N; k++)
      c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

    transforms to

```
for (int i = 0; i < N; i++)
  for (int k = 0; k < N; k++)
    for (int j = 0; j < N; j++)
      c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

    where the j and k loops have been swapped
  - `-floop-unroll-and-jam` unrolls outer loops and fuses the inner loops
  - `-fpeel-loops` peels loops that don't roll much. Additionally unrolls small loops.
  - `-fpredictive-commoning` reuses computations (especially memory loads and stores) performed in previous iterations of loops.
  - `-fsplit-loops` splits loops if there's a condition that's always true for one side of the iteration space, and false for the other.
  - `-fvect-cost-model=dynamic` allows for automatic-vectorization when a considered loop has a number of iterations that will likely execute faster than when executing the original scalar loop.
  - `-fversion-loops-for-strides` if a loop iterates over an array with a variable stride, create another version of the loop that assumes the stride is always one. For example:

```
for (int i = 0; i < n; ++i)
  x[i * stride] = ...;
```

    becomes

```
if (stride == 1)
  for (int i = 0; i < n; ++i)
    x[i] = ...;
```

```
else
  for (int i = 0; i < n; ++i)
    x[i * stride] = ...;
```

This is particularly useful for assumed-shape arrays in Fortran where (for example) it allows better vectorization assuming contiguous accesses.

- `-Ofast`
  - `-ffast-math` enables non-IEEE floating point operations, but could lead to incorrect values in a program that depends on IEEE specifications.
  - `-fallow-store-data-races` provides optimizations that can introduce data-races. Can be used safely in single-threaded applications.
  - (Fortran only) `-fstack-arrays` allocates all local-arrays from stack memory.

2.3 Manual Optimization

2.3.1 Memory Caching

- Spatial Locality: Fortran is column-major in its multidimensional arrays, so to take advantage of compiler prefetching and quicker memory access, the left-most index should be associated with the innermost loop (so each column is traversed instead of each row)
- Temporal Locality:

2.3.2 Miscillaneous

- minimize indexing

3 Results

4 Discussion

5 Conclusion

References

[1] FLang. *Stack arrays pass*. URL: `https://flang.llvm.org/docs/fstack-arrays.html`.

[2] GNU. *Options That Control Optimization*. URL: `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.

[3] Ryad El Khatib. *General Fortran optimizations guide*. 2019. URL: `https://www.umr-cnrm.fr/gmapdoc/IMG/pdf/general_fortran_optimization_guide_manual.pdf`.