# Project #5
# Vectorized Array Multiplication and Reduction

By Michael Rose

Due: May 17th

Written for: CS475 Spring 2016

Oregon State University

## Table of Contents

## Code

### Non-Vectorized Multiplication

```
void
Mul(float *a, float *b, float *c, int len) {
	for (int i = 0; i < len; i++) {
		c[i] = a[i] * b[i];
	}
}
```

### Non-Vectorized Multiplication and Reduction

```
float
MulSum(float *a, float*b, int len) {
	float sum = 0.;
	for (int i = 0; i < len; i++) {
		sum += a[i] * b[i];
	}
	return sum;
}
```

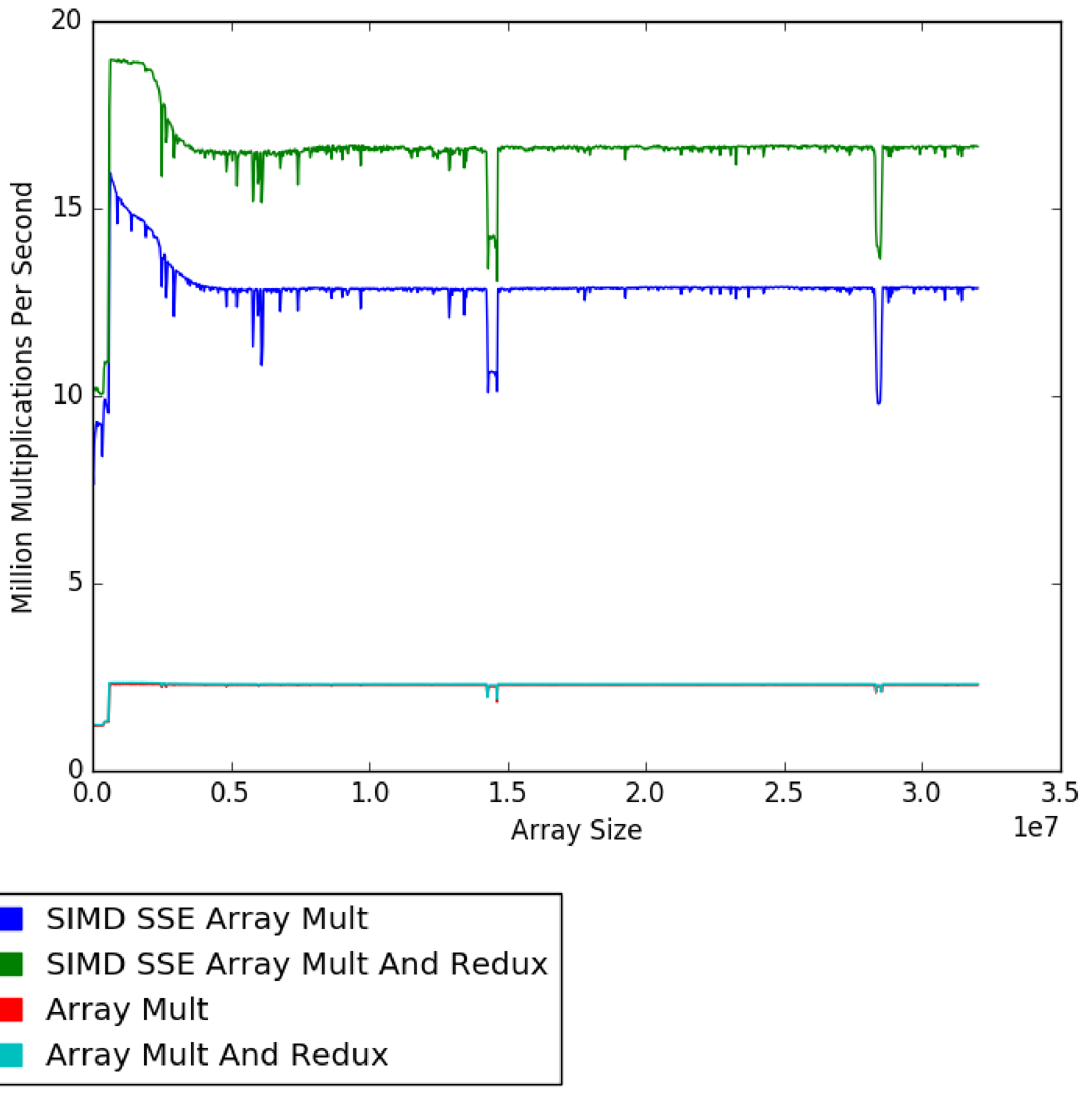
## What I did

I ran my code on flip. I combined the asm code into my own cpp file and just had 1 file to run all 4 functions.

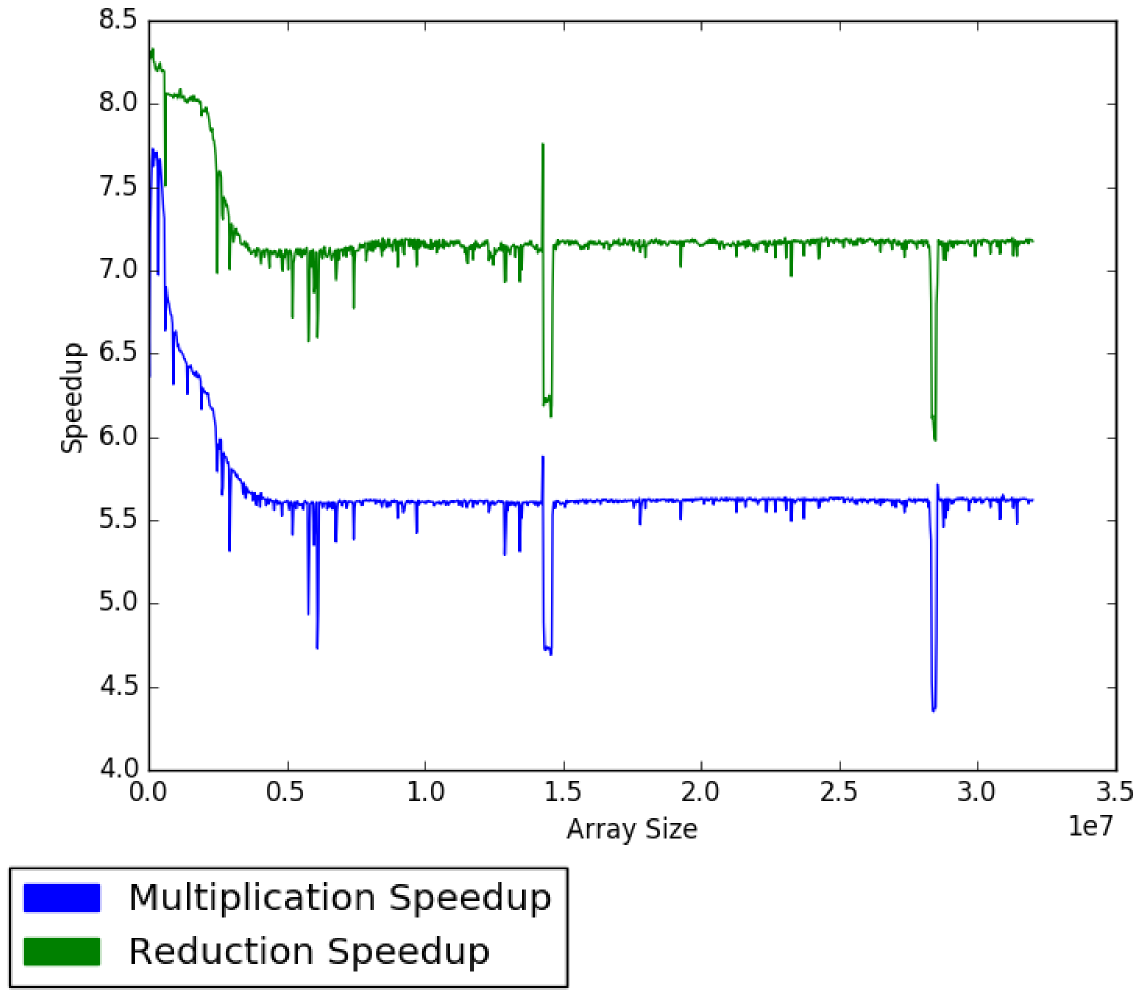
The main function creates 3 arrays of size ARRSIZE (default 32 million). It has a for loop that runs 1000 times. The loop increments by (ARRSIZE-MINSIZE)/1000 so that no matter what the max size is, it has 1000 data points. Within the loop, there is another loop to get an average rather than just doing a single run. The default is 10 runs for each array size. My simple functions can be seen in the previous sections. I set up my code to run each function on the same array size before incrementing. This means that if flip is busy, it will probably affect all the data points at that array size, but it won't create a random dip in just one of the functions. In my results this can actually be seen.

## Results

### Graph: MegaMults vs. Array Size



## Graph: Speedup



## Table: Speedup

| | Size | Non-SIMD | SIMD | Speedup |
|---|---|---|---|---|
| Smallest Multiplication Speedup | 28416112 | 2.252 | 9.796 | 4.349 |
| Largest Multiplication Speedup | 160995 | 1.204 | 9.310 | 7.730 |
| Smallest Reduction Speedup | 28480110 | 2.285 | 13.659 | 5.977 |
| Largest Reduction Speedup | 160995 | 1.226 | 10.213 | 8.331 |

Average Multiplication Speedup: 5.6799293598

Average Redux Speedup: 7.20445212946

## Analysis

Pyplot created the scale for the x-axis so it shows 3.5e7 instead of 35,000,000.

### Graph: MegaMults vs. Array Size

The cyan and red lines are virtually on top of each other because the two for loops are basically doing the same thing. The difference is where and how they are storing the result.  SIMD SSE array multiplication has a speedup of around 6x and SIMD SSE array multiplication and reduction has a speedup of around 8x. We get more than 4x speedup because 4x speedup is the minimum we should expect. The SIMD SSE code keeps as much data in the registers as possible, rather than moving data to and from the stack. This creates even more speed up.

### Graph and Table: Speedup

The 2nd graph shows the speedup for each value. It is clear that the speedup for low values is great, but the important speedup has more to do with large arrays. It is very consistent after array sizes of around 500,000. Looking at the table, we can see that early on the speedup reaches a max speedup of 7.730 for multiplication and 8.331 for reduction. The smallest speedup seems to be more affected by the other processes running on flip, the minimum value occurs at the last large trench, but there are a few different trenches along the “line” of the average speedup values. Those averages can be seen just below the table.