OPTIMIZING MATRIX MULTIPLICATION

PROFILING FOR A BASELINE

Using the naive approach for matrix multiplication, written in C. We profile using gprof to obtain a baseline.

COMPILER OPTIMZATION

Now we add gcc’s already available optimization options to the mix to improve the speeds further.

ALGORITHM OPTIMIZATION

Just changing the loop order decreases the exec time by almost 5x for 2048x2048 matrices due to how the cache behaves. Using Block execution and straussen’s algorithms can be implemented but haven't been discussed

PARELLIZATION-SIMD & SIMT

Using SIMD instructions, we can optimize the lloops even further. Using CUDA on an NVIDIA GPU, we can use multiple threads for the simultaneous execution

BASELINE MEASURING

- First we write the naive implementation of matrix multiplication in C:

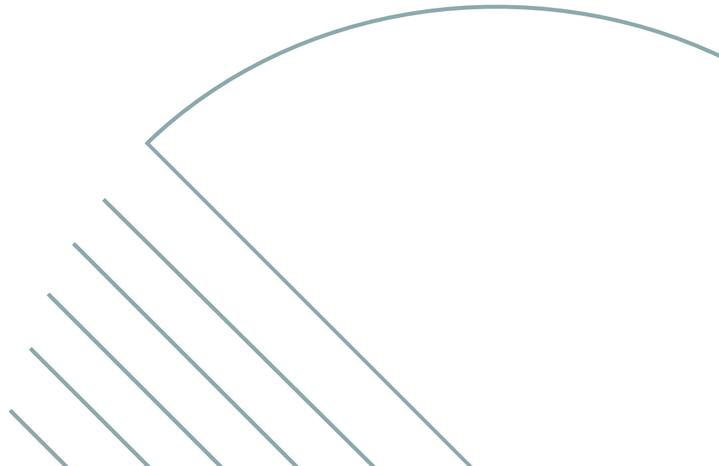
```c
for(int i = 0; i < n; i++){
    for(int j = 0; j < n; j++){
        for(int k = 0; k < n; k++){
            c[i][j] += a[i][k] * b[k][j];
```

- We then compile the program with gcc with profiling enabled, using the `-pg` option enabled.

- We then use the gprof tool to get the analysis of each function call's performance.

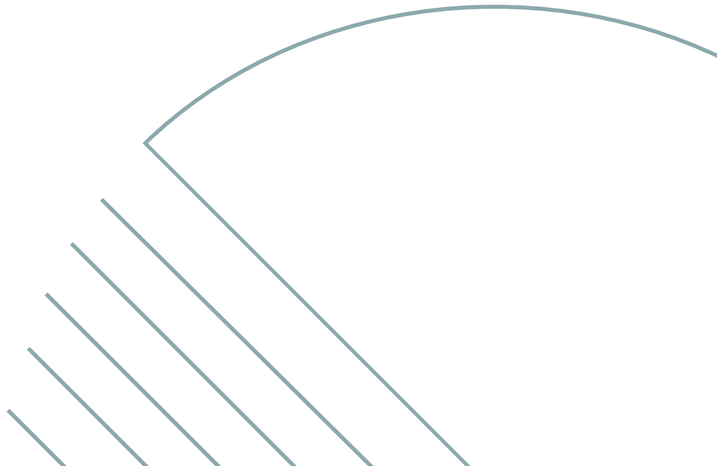
COMPILER OPTIMIZATIONS

- We can enable gcc's inbuilt 3 level optimizations using the -O flags.
- The -O2 flags enables all optimizations that do not involve a space-speed tradeoff
- Using the -O1 flag enables basic optimizations that do not increase compilation time significantly and reduce code size, removing unnecessary assignments and loops.
- The -O3 flag enables all the previous optimizations and adds even more aggressive optimization flags. Sometimes might result in very hard-to-debug problems.



ALGORITHM OPTIMIZATION - CACHE HITS

- The i-j-k loop, described in the naive approach is a column-major approach, with a time complexity of O(n3).
- Accessing a large array (like 2048*2048 floats in size) in large strides, as we are doing in the i-j-k loop leads to cache misses, as the contiguous values loaded in the cache are not used.
- Changing the loop order to i-k-j changes the execution in a row-major configuration in the innermost loop, reducing the number of cache misses by an order of ~n



ALGORITHM OPTIMIZATIONS: OTHER ALGORITHMS

- To prevent cache misses in even larger matrices, we use the “divide and conquer” approach, by dividing the matrices into blocks that fit in the CPU’s L1 cache.
- This heavily depends on the architecture of the CPU itself and therefore the optimization may not be universally effective

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix},$$

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

ALGORITHM OPTIMIZATIONS: SUB-CUBIC ALGORITHMS

- The first sub-cubic matrix multiplication algorithm was the ‘Strassen’s Algorithm’, with the time complexity of O(n^2.807). For a 2x2 matrix, it replaces 8 multiplications with 7, which are the most time consuming operation.

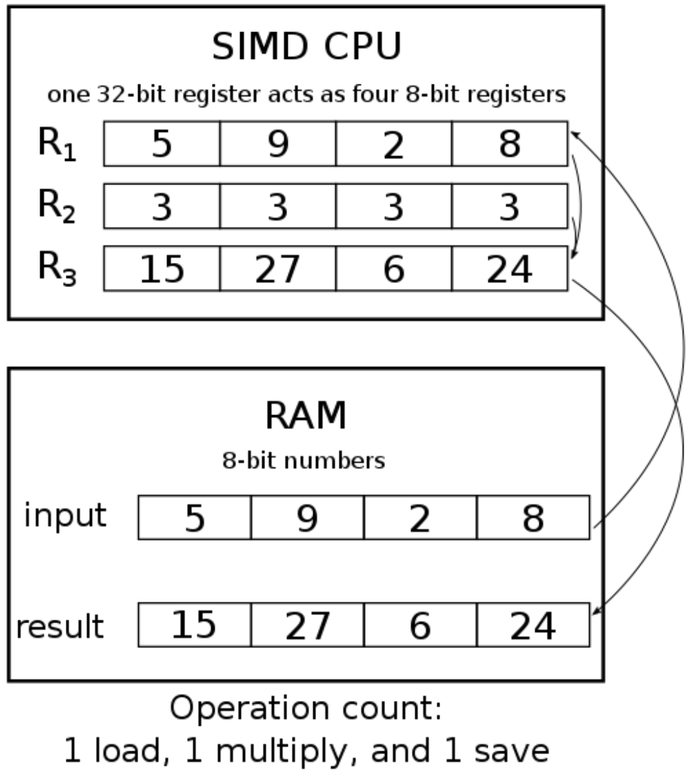
- In 2022, google’s DeepMind introduced AlphaTensor, a neural network which single-player game analogy to invent and rediscover some very efficient matrix multiplcation

- As a tradeoff, it adds a few additions and subtractions, which are relatively faster than multiplication, lending the algorithm its sub-cubic time complexity.

- The best "practical" (explicit low-rank decomposition of a matrix multiplication tensor) algorithm found ran in O(n2.778). Finding low-rank decompositions of such tensors (and beyond) is NP-hard; optimal multiplication even for 3x3 matrices remains unknown. (Wikipedia)

PARALLELIZATION: SIMD

- SIMD, which stands for Single Instruction Multiple Data, uses the same number of CPU clocks to apply operations on multiple data simultaneously in a single thread.



Operation count:
1 load, 1 multiply, and 1 save

- Most modern CPUs have SIMD instructions available and therefore can be used to optimize our algorithm considerably

- We can use OpenMP's 'pragma' directive to easily implement SIMD in our specific case.
- #pragma omp parallel for private(i,j,k) shared(a,b,c)

PARALLELIZATION : CUDA AND SIMT

- We can use Nvidia's 'nvcc' compiler to compile CUDA programs with the extension '.cu'

- We define a kernel, allocate host and device memory, copy inputs to device, calculate results and then copy results back to host.

```
__global__ void matmul(float* a, float* b, float* c, int n) {
    int row = blockIdx.y*blockDim.y+threadIdx.y;
    int col = blockIdx.x*blockDim.x+threadIdx.x;

    float tmpSum = 0;

    if (row < n && col < n) {
        // each thread computes one element of the block sub-matrix
        for (int i = 0; i < n; i++) {
            tmpSum += a[row * n + i] * b[i * n + col];
        }
    }
    c[row * n + col] = tmpSum;
}
```

```
float *a, *b, *c, *d_a, *d_b, *d_c;

a = (float *) malloc (n*n*sizeof(float));
b = (float *) malloc (n*n*sizeof(float));
c = (float *) malloc (n*n*sizeof(float));
// initialize arrays, a and b with random floats and c with zeros
populate(a, 1); populate(b, 1); populate(c, 0)

// allocate device memory
cudaMalloc((void**)&d_a, sizeof(float)*n*n);
cudaMalloc((void**)&d_b, sizeof(float)*n*n);
cudaMalloc((void**)&d_c, sizeof(float)*n*n);
// copy input to device
cudaMemcpy(d_a, a, sizeof(float)*n*n, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(float)*n*n, cudaMemcpyHostToDevice);
// calculate blocks per grid and threads per block
dim3 threadsPerBlock(n, n);
dim3 blocksPerGrid(1, 1);
if (n*n > 512){
    threadsPerBlock.x = 512;
    threadsPerBlock.y = 512;
    blocksPerGrid.x = ceil(double(N)/double(threadsPerBlock.x));
    blocksPerGrid.y = ceil(double(N)/double(threadsPerBlock.y));
}
matmul<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, n); // Call the kernel
cudaMemcpy(c, d_c, sizeof(float)*n*n, cudaMemcpyDeviceToHost);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
```