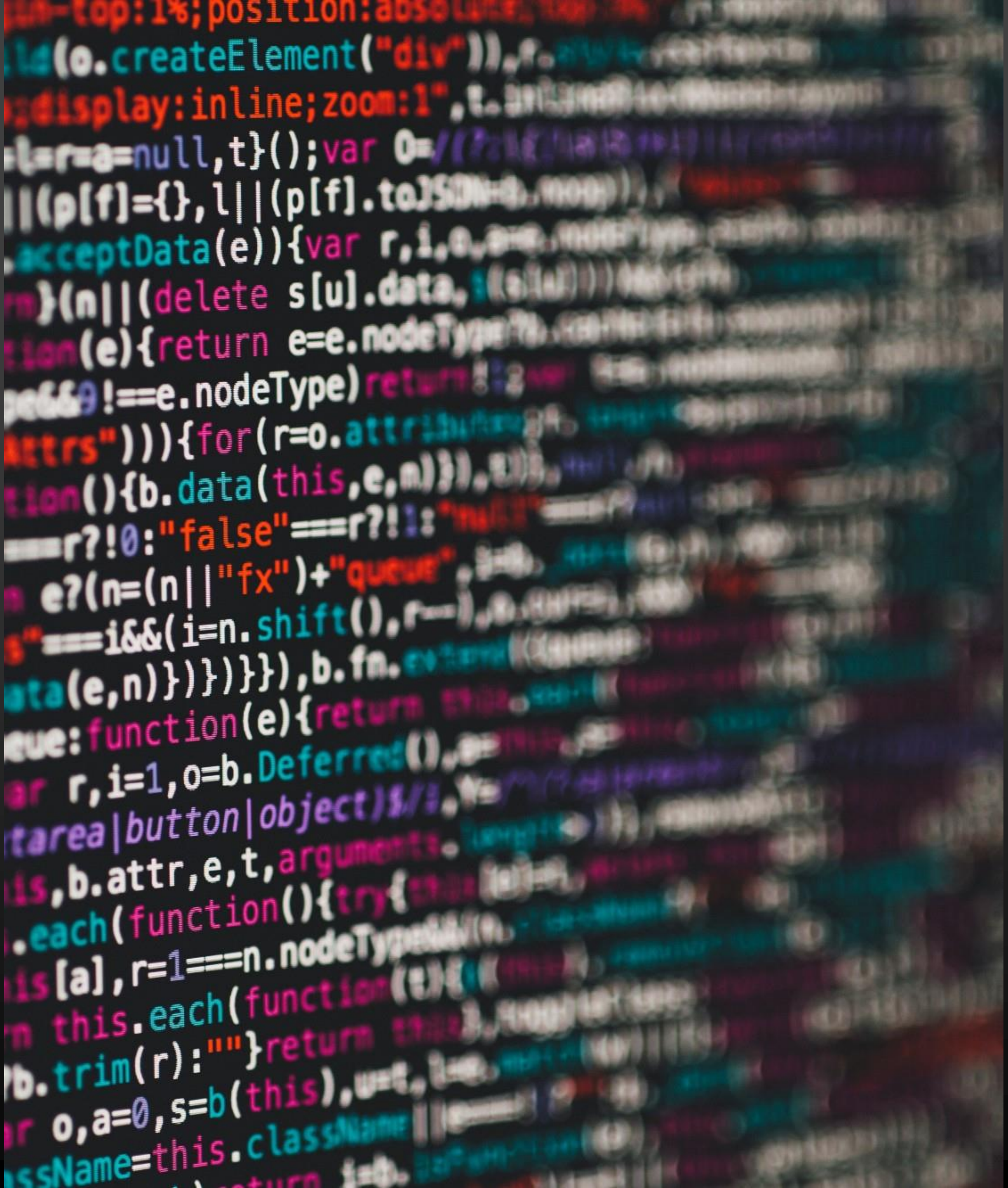
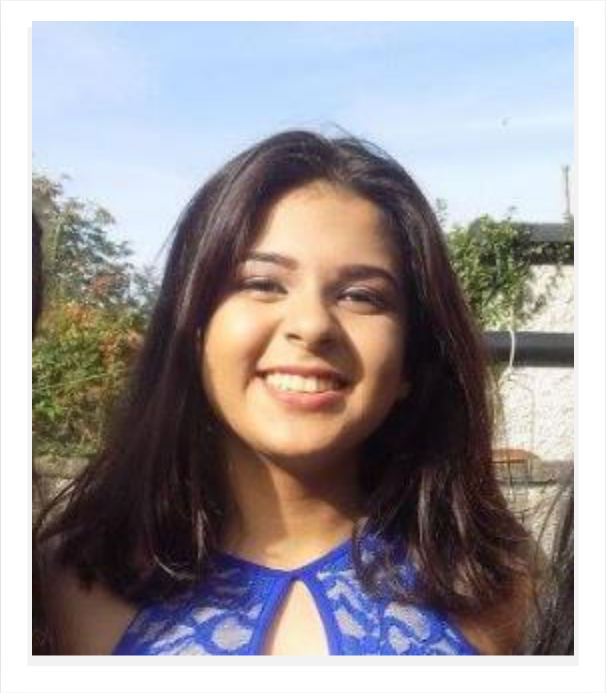
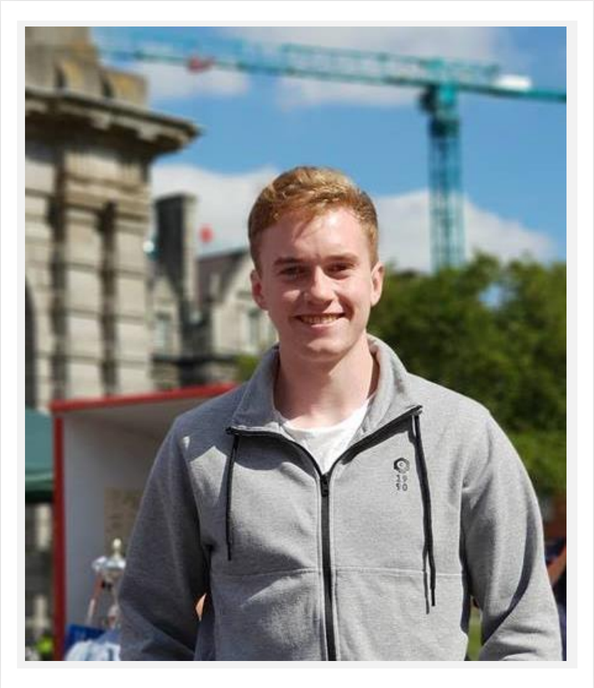
Parallel multichannel multikernel convolution



The Team





Original Function as provided

```c
/* the fast version of matmul written by the team */
void team_conv(int16_t *** image, int16_t **** kernels, float *** output,
               int width, int height, int nchannels, int nkernels,
               int kernel_order)
{
  int h, w, x, y, c, m;
  for ( m = 0; m < nkernels; m++ ) {
    for ( w = 0; w < width; w++ ) {
      for ( h = 0; h < height; h++ ) {
        double sum = 0.0;
        for ( c = 0; c < nchannels; c++ ) {
          for ( x = 0; x < kernel_order; x++) {
            for ( y = 0; y < kernel_order; y++ ) {
              sum += (double) image[w+x][h+y][c] * (double) kernels[m][c][x][y];
            }
          }
          output[m][w][h] = (float) sum;
        }
      }
    }
  }
}
```

Adding in a parallelization on three for loops

```
/* the fast version of matmul written by the team */
void team_conv(int16_t *** image, int16_t **** kernels, float *** output,
               int width, int height, int nchannels, int nkernels,
               int kernel_order)
{
  int h, w, x, y, c, m;
  #pragma omp parallel for collapse(3)
  for ( m = 0; m < nkernels; m++ ) {
    for ( w = 0; w < width; w++ ) {
      for ( h = 0; h < height; h++ ) {
        double sum = 0.0;
        for ( c = 0; c < nchannels; c++ ) {
          for ( x = 0; x < kernel_order; x++) {
            for ( y = 0; y < kernel_order; y++ ) {
              sum += (double) image[w+x][h+y][c] * (double) kernels[m][c][x][y];
            }
          }
          output[m][w][h] = (float) sum;
        }
      }
    }
  }
}
```

Width	Height	Kernel_ Order	Number_Of_ Channels	Kernel_ Number		No Improvement	Parallelized collapse(1)		Parallelized collapse(2)		Parallelized collapse(3)		
16	16	5	1024	128		2864872	164775	17	162546	16	163035	15	
32	32	3	512	256		4137490	242200	17	254254	14	249677	14	
64	64	1	256	64		442745	28149	15	29390	12	28495	12	
128	128	7	128	128		32516436	1774827	18	1574541	20	1582756	20	
256	256	5	64	64		19271368	1003890	19	1019410	18	971688	19	
512	512	7	32	32		32403120	2352950	13	1585599	20	1618731	20	

Step 1: Parallelize by threading to #pragma omp parallel for collapse(3)

Run this on Stoker. Time results as follows: (in microseconds)

Step 2: Try to Parallelize by threading to #pragma omp parallel for collapse(6)

This didn't work….
We had to add in some complicated math to reorder stuff
Well, actually it did work, to a degree for really large inputs

Run this on Stoker. Time results as follows: (in microseconds)

Step 3: Try to Vectorize using _m128d

Again, this didn't work….
Well, again, it did, to a degree for really large numbers,
but for smaller ones, on average there was a 1.2 – 2x speedup.

Run this on Stoker. Time results as follows: (in microseconds)

Adding in a Vectorization (as from the notes)

```
void team_conv(int16_t ***  image, int16_t ****  kernels, float ***  output,
               int width, int height, int nchannels, int nkernels,
               int kernel_order)
{
  int h, w, x, y, c, m;
  double**** newKernels = new_empty_4d_matrix_double(nkernels, kernel_order, kernel_order, nchannels);
  #pragma omp parallel for collapse(4)
  for (int i = 0; i < nkernels; i++)
  {
    for (int j = 0; j < nchannels; j++)
    {
      for (int k = 0; k < kernel_order; k++)
      {
        for(int l = 0; l < kernel_order; l++)
        {
          newKernels[i][k][l][j] = (double)kernels[i][j][k][l];
        }
      }
    }
  }
  #pragma omp parallel for collapse(1)
  for ( m = 0; m < nkernels; m++ ) {

    for ( w = 0; w < width; w++ ) {
      for ( h = 0; h < height; h++ ) {
        double sum = 0.0;
        for ( x = 0; x < kernel_order; x++) {
          for ( y = 0; y < kernel_order; y++ ) {
            #pragma omp simd safelen(4)
            for(c = 0; c < nchannels; c++) {
              sum += (double)image[w+x][h+y][c] * newKernels[m][x][y][c];
            }
          }
          output[m][w][h] = (float) sum;
        }
      }
    }
  }
```

Step 4: Add in some real vectorization (we referred to the notes(Lecture 7) for this!)

Run this on Stoker. Time results as follows: (in microseconds)

Width	Height	Kernel_ Order	Number_Of_ Channels	Kernel_ Number		No Improvement	Parallelized collapse(3)		Vectorization collapse(3)		Vectorization collapse(1)		
16	16	5	1024	128		2864872	163035	15	110574	29	106550	26	
32	32	3	512	256		4137490	249677	14	146560	29	146553	25	
64	64	1	256	64		442745	28495	12	8816	44	9284	41	
128	128	7	128	128		1383479	1605436	20	1086641	30	1083797	29	
256	256	5	64	64		884024	987276	17	709312	26	661981	28	
512	512	7	32	32		2156709	1424520	22	1045832	30	1186200	27	

Live Demo