Fast brute force matching of 128D points

Topics in X: GPU Programming

Mårten Björkman

Robotics, Perception and Learning Lab
KTH Royal Institute of Technology

Matching problem

- Assume we have 16K 128D feature points that we like to match against another 16K feature points.
- With each feature normalised to a length equal to 1, matching scores are computed as dot products.
- In total, we would need 16K*16K*128 = 32G multiply and adds.
- In theory, we cannot do that faster than 5.2 ms in a RTX 2080 Ti.
- How close can be get?

First Naive CPU version

As a baseline with begin with a naive CPU version.

```
void MatchCPU1(float *pts1, float *pts2, float *score, int *index) {
  std::memset(score, 0, sizeof(float)*NPTS);
  for (int p1=0; p1<NPTS; p1++) {
    for (int p2=0; p2<NPTS; p2++) {
      float score = 0.0f;
      for (int d=0; d<NDIM; d++)
        score += pts1[p1*NDIM + d]*pts2[p2*NDIM + d];
      if (score>score[p1]) {
        score[p1] = score;
        index[p1] = p2;
      }
    }
  }
}
```

- Time consumption: 34.6 s, or 1.89 Gflops on a Xeon Gold 5118
- Problem: For each outer iteration, all `p2` points are read, which leads to the L2 cache being repeatably corrupted.
- Solution: Divide `p1` and `p2` points into groups, so that each such group fits the L2 cache.

Second CPU version

```
void MatchCPU2(float *pts1, float *pts2, float *score, int *index) {
  #define BSIZ  256
  std::memset(score, 0, sizeof(float)*NPTS);
  for (int b1=0; b1<NPTS; b1+=BSIZ)
    for (int b2=0; b2<NPTS; b2+=BSIZ)
      for (int p1=b1; p1<b1 + BSIZ; p1++) {
        float *pt1 = &pts1[p1*NDIM];
        for (int p2=b2; p2<b2 + BSIZ; p2++) {
          float *pt2 = &pts2[p2*NDIM];
          __m256 score8 = _mm256_setzero_ps();
          for (int d=0; d<NDIM; d+=8) {
            __m256 v1 = _mm256_load_ps(pt1 + d);
            __m256 v2 = _mm256_load_ps(pt2 + d);
            score8 = _mm256_fmadd_ps(v1, v2, score8);
          }
          score8 = _mm256_add_ps(score8, _mm256_permute2f128_ps(score8, score8, 1));
          score8 = _mm256_hadd_ps(score8, score8);
          float score = _mm256_cvtss_f32(_mm256_hadd_ps(score8, score8));
          if (score>score[p1]) {
            score[p1] = score;
            index[p1] = p2;
          }
        }
      }
}
```

- Modification: Divide points `p1` and `p2` into groups of 256 points each and use SIMD intrinsics to work on vectors of 8 floats each.
- Time consumption: 3.06 s, or 21.4 Gflops, x11.3 vs Naive CPU

Third CPU version

```
    void MatchCPU3(float *pts1, float *pts2, float *score, int *index) {
      #define BSIZ  256
      std::memset(score, 0, sizeof(float)*NPTS);
#pragma omp parallel for
      for (int b1=0; b1<NPTS; b1+=BSIZ)
        for (int b2=0; b2<NPTS; b2+=BSIZ)
          for (int p1=b1; p1<b1 + BSIZ; p1++) {
            float *pt1 = &pts1[p1*NDIM];
            for (int p2=b2; p2<b2 + BSIZ; p2++) {
              float *pt2 = &pts2[p2*NDIM];
              ...
            }
          }
    }
```

- The Xeon Gold 5118 has 12 cores. Why not use them?
- Simple motification: Just add a pragma and compile with OpenMP.
- Time consumption: 185 ms, or 354 Gflops, x188 vs Naive CPU
- How does this compare against a similar priced GPU?

First GPU version – naive

```
__global__ void MatchCPU1(float *pts1, float *pts2, float *score, int *index) {
  int p1 = threadIdx.x + 128*blockIdx.x;
  float max_score = 0.0f;
  int index = -1;
  for (int p2=0;p2<NPTS;p2++) {
    float score = 0.0f;
    for (int d=0;d<NDIM;d++)
      score += pts1[p1*NDIM + d]*pts2[p2*NDIM + d];
    if (score>max_score) {
      max_score = score;
      index = p2;
    }
  }
    score[p1] = max_score;
  index[p1] = index;
}
```

- Each thread keeps a `p1` point that is matched to all `p2` points.
- Time consumption: 642 ms, or 102 Gflops, x54 vs Naive CPU on a RTX 2080 Ti GPU
- Problem: Not enough threads to fill up all cores with enough work and too much pressure on global memory
- Solution: Use 256 threads to match 16 `p1` and 16 `p2` points at the time, with buffering in shared memory

Second GPU version – shared buffers

```
__global__ void MatchGPU2(float *pts1, float *pts2, float *score, int *index)
{
  __shared__ float buffer1[16*NDIM], buffer2[16*NDIM], scores[16*16];
  float max_score = 0.0f;
  int index = -1;
  int tx = threadIdx.x, ty = threadIdx.y;
  int idx = tx + 16*ty, bp1 = 16*blockIdx.x;

  for (int d=tx;d<NDIM;d+=16)
    buffer1[ty*NDIM + d] = pts1[(bp1 + ty)*NDIM + d];      // read 16 p1 points

  for (int bp2=0;bp2<NPTS;bp2+=16) {
    for (int d=tx;d<NDIM;d+=16)
      buffer2[ty*NDIM + d] = pts2[(bp2 + ty)*NDIM + d];    // read 16 p2 points
    __syncthreads();

    float score = 0.0f;
    for (int d=0;d<NDIM;d++)
      score += buffer1[tx*NDIM + d]*buffer2[ty*NDIM + d]; // compute matching scores
    scores[idx] = score;
    __syncthreads();

    if (ty==0)
      for (int i=0;i<16;i++)
        if (scores[i*16 + tx]>max_score) {
          max_score = scores[i*16 + tx];                   // update best matches
          index = bp2 + i;
        }
  }
  ...
}
```

Second GPU version – shared buffers

```
__global__ void MatchGPU2(float *pts1, float *pts2, float *score, int *index)
{
  __share_ float buffer1[16*NDIM], buffer2[16*NDIM], scores[16*16];
  float max_score = 0.0f;
  int index = -1;

  ...

  if (ty==0) {
    score[bp1 + tx] = max_score;                          // store in device memory
    index[bp1 + tx] = index;
  }
}
```

- SM occupancy up from 20% to 72%, with maximum at 75%.
- Time consumption: 148 ms, or 443 Gflops, x4.3 vs Naive GPU
- Problem: Too many requests to shared memory due to bank conflicts, in most critical loop that computes matching scores
- Solution: Add one element per point of padding to the `p1` buffer

Third GPU version – shared padding

```
__global__ void MatchGPU3(float *pts1, float *pts2, float *score, int *index)
{
  __shared__ float buffer1[16*(NDIM + 1)], buffer2[16*NDIM], scores[16*16];
  float max_score = 0.0f;
  int index = -1;
  ...
  for (int d=tx;d<NDIM;d+=16)
    buffer1[ty*(NDIM + 1) + d] = pts1[(bp1 + ty)*NDIM + d];      // read 16 p1 points

  for (int bp2=0;bp2<NPTS;bp2+=16) {
    ...                                                           // read 16 p2 points
    float score = 0.0f;
    for (int d=0;d<NDIM;d++)
      score += buffer1[tx*(NDIM + 1) + d]*buffer2[ty*NDIM + d]; // compute matching scores
    scores[idx] = score;
    __syncthreads();
    ...                                                           // update best matches
  }
  ...                                                             // store in device memory
}
```

Third GPU version – shared padding

- Shared memory requests down from 9.1G to 1.6G.
- Time consumption: 31.9 ms, or 2050 Gflops, x20.1 vs Naive GPU
- Problem: Compute utilisation is at 72%, but this is dominated by the LSU (load store unit) that has a utilisation of 76%, while FMA (multipy-add unit) only has one of 15%
- We want the FMA utilisation to be as high as possible, because this is where we compute what we want
- Solution: Use float4 instead of float, which means that 4 floats are loaded and stored with the same shared memory request

Fourth GPU version – float4

```
__global__ void MatchGPU4(float *pts1, float *pts2, float *score, int *index)
{
  __shared__ float4 buffer1[16*(NDIM/4 + 1)], buffer2[16*NDIM/4];
  ...
  for (int d=tx;d<NDIM/4;d+=16)                                  // read 16 p1 points
    buffer1[ty*(NDIM/4 + 1) + d] = ((float4*)pts1)[(bp1 + ty)*(NDIM/4) + d];

  for (int bp2=0;bp2<NPTS;bp2+=16) {
    for (int d=tx;d<NDIM/4;d+=16)                                // read 16 p2 points
      buffer2[ty*NDIM/4 + d] = ((float4*)pts2)[(bp2 + ty)*(NDIM/4) + d];
    __syncthreads();

    float score = 0.0f;
    for (int d=0;d<NDIM/4;d++) {                                 // compute matching scores
      float4 v1 = buffer1[tx*(NDIM/4 + 1) + d];
      float4 v2 = buffer2[ty*(NDIM/4) + d];
      score += v1.x*v2.x;
      score += v1.y*v2.y;
      score += v1.z*v2.z;
      score += v1.w*v2.w;
    }
    scores[idx] = score;
    __syncthreads();

    ...                                                          // update best matches
  }
  ...                                                            // store in device memory
}
```

Fourth GPU version – float4

- LSU utilisation is down from 76% to 33%, while FMA utilisation is up from 15% to 16%
- Time consumption: 29.8 ms, or 2200 Gflops, x21.5 vs Naive GPU
- Problem: The pressure on LSU has decreased, but it still takes time to load the actual data
- Solution: Let each thread do multiple matches at the time and use registers for caching of `p1`, instead of always loading from shared

Fifth GPU version – multiple matches

```
__global__ void MatchGPU5(float *pts1, float *pts2, float *score, int *index)
{
  __shared__ float4 buffer1[16*(NDIM/4 + 1)], buffer2[16*NDIM/4];
  __shared__ float scores[16*16];
  ...                                                     // read 16 p1 points

  for (int bp2=0;bp2<NPTS;bp2+=16) {
    ...                                                   // read 16 p2 points

    if (ty<4) {
      float score[4];
      for (int dy=0;dy<4;dy++)
        score[dy] = 0.0f;
      for (int d=0;d<NDIM/4;d++) {                        // only read p1 data ones
        float4 v1 = buffer1[tx*(NDIM/4 + 1) + d];
        for (int dy=0;dy<4;dy++) {                        // compute matching score
          float4 v2 = buffer2[(4*ty + dy)*(NDIM/4) + d];
          score[dy] += v1.x*v2.x;                         // use p1 data four times
          score[dy] += v1.y*v2.y;
          score[dy] += v1.z*v2.z;
          score[dy] += v1.w*v2.w;
        }
      }
      for (int dy=0;dy<4;dy++)
        scores[tx + 16*(4*ty + dy)] = score[dy];
    }
    __syncthreads();

    ...                                                   // update best matches
  }
  ...                                                     // store in device memory
}
```

Fifth GPU version – multiple matching

- LSU utilisation is up from 33% to 39%, but FMA utilisation is also up from 16% to 26%, which is more important
- Time consumption: 17.1 ms, or 3822 Gflops, x37.4 vs Naive GPU
- Problem: Unnecessary shared stores in the loop over `p2` points
- Solution: Store best matches and indices in registers

Sixth GPU version – no shared scores

```
__global__ void MatchGPU6(float *pts1, float *pts2, float *score, int *index)
{
  __shared__ float4 buffer1[16*(NDIM/4 + 1)], buffer2[16*NDIM/4];
  ...                                                          // read 16 p1 points

  for (int bp2=0;bp2<NPTS;bp2+=16) {
    ...                                                        // read 16 p2 points

    if (ty<4) {
      float score[4];
      ...                                                      // compute matching scores

      for (int dy=0;dy<4;dy++) {
        if (score[dy]>max_score) {                             // update best matches
          max_score = score[dy];
          index = bp2 + 4*ty + dy;
        }
      }
    }
    __syncthreads();
  }

  float *scores = (float*)buffer1;                             // reuse buffer1 for scores
  int *indices = (int*)&scores[16*4];
  if (ty<4) {
    scores[ty*16 + tx] = max_score;                            // store matches in shared
    indices[ty*16 + tx] = index;
  }
  __syncthreads();

  ...                                                          // store in device memory
}
```

Sixth GPU version – no shared scores

- LSU utilisation is down from 39% to 38%, but FMA utilisation is up from 26% to 27%, which is a minor change
- Time consumption: 16.3 ms, or 4008 Gflops, x39.3 vs Naive GPU
- Problem: As many as 3/4 threads only load, but never compute. Padding can be avoided with circulant buffer.
- Solution: Change to 32x32 matches per block, but with 32x8 threads. Use circulant matrix for `p1` points instead of padding.

Seventh version – larger buffers

```
  __global__ void MatchGPU7(float *pts1, float *pts2, float *score, int *index)
  {
    __shared__ float4 buffer1[16*NDIM/4], buffer2[16*NDIM/4];
    ...

    for (int j=ty;j<32;j+=8)                                 // read 32 p1 points
      buffer1[j*NDIM/4 + (tx + j)%(NDIM/4)] = ((float4*)pts1)[(bp1 + j)*(NDIM/4) + tx];
    ...

    for (int bp2=0;bp2<NPTS;bp2+=32) {
      for (int j=ty;j<32;j+=8)                               // read 32 p2 points
        buffer2[j*NDIM/4 + tx] = ((float4*)pts2)[(bp2 + j)*(NDIM/4) + tx];
      __syncthreads();
      ...                                                    // compute matching scores

      ...                                                    // update best matches
    }
    __syncthreads();
  }
  ...                                                        // store matches in shared

  ...                                                        // store in device memory
}
```

Seventh version – larger buffers

- LSU utilisation is up from 38% to 39%, while FMA utilisation is up from 27% to 33%, which is more important
- Time consumption: 14.8 ms, or 4428 Gflops, x43.3 vs Naive GPU
- Problem: Stalls dominated by MIO Throttle (shared memory) in the inner loop (8.3 cycles per instruction)
- Solution: Let each thread find matches for two features at the time

Eighth version – multiple features

```
__global__ void MatchGPU8(float *pts1, float *pts2, float *score, int *index)
{
  __shared__ float4 buffer1[16*NDIM/4], buffer2[16*NDIM/4];
  ...                                                       // read 32 p1 points

  for (int bp2=0;bp2<NPTS;bp2+=32) {
    ...                                                     // read 32 p2 points
    float score[4][2];
    ...
    for (int d=0;d<NDIM/4;d++) {
      float4 v1[2];
      for (int dx=0;dx<2;dx++)
         v1[dx] = buffer1[(16*dx + tx%16)*(NDIM/4) + (d + 16*dx + tx%16)%(NDIM/4)];
      for (int dy=0;dy<4;dy++) {
        float4 v2 = buffer2[(4*(2*ty + tx/2) + dy)*(NDIM/4) + d];
        for (int dx=0;dx<2;dx++) {                          // compute matching scores
          score[dy][dx] += v1[dx].x*v2.x; score[dy][dx] += v1[dx].y*v2.y;
          score[dy][dx] += v1[dx].z*v2.z; score[dy][dx] += v1[dx].w*v2.w;
        }
      }

      ...                                                    // update best matches
    }
    __syncthreads();
  }
  ...                                                        // store matches in shared

  ...                                                        // store in device memory
}
```

Eighth version – multiple features

- LSU utilisation is down from 39% to 35%, while FMA utilisation is up from 33% to 47%, which is more important
- Time consumption: 10.5 ms, or 6224 Gflops, x61.0 vs Naive GPU
- Problem: Stalls dominated by Barriers when `p2` points are loaded (6.3 cycles per instruction)
- Solution: Have a cool beer and try to forget about it!

Summary of GPU optimisations

	Time	Performance	Modification
MatchGPU1	642.0 ms	102 Gflops	Initial naive version
MatchGPU2	148.0 ms	443 Gflops	Shared memory buffering
MatchGPU3	31.9 ms	2051 Gflops	Padded shared buffer
MatchGPU4	29.8 ms	2200 Gflops	Loads using float4
MatchGPU5	17.1 ms	3822 Gflops	Four matches per thread
MatchGPU6	16.4 ms	4008 Gflops	Delayed shared stores
MatchGPU7	14.8 ms	4428 Gflops	Larger windows
MatchGPU8	10.5 ms	6224 Gflops	Two features per thread

Most important changes

- Reduce global loads through buffering in shared memory
- Add padding to shared buffers to eliminate bank conflicts
- Make threads do more by reusing shared data already read