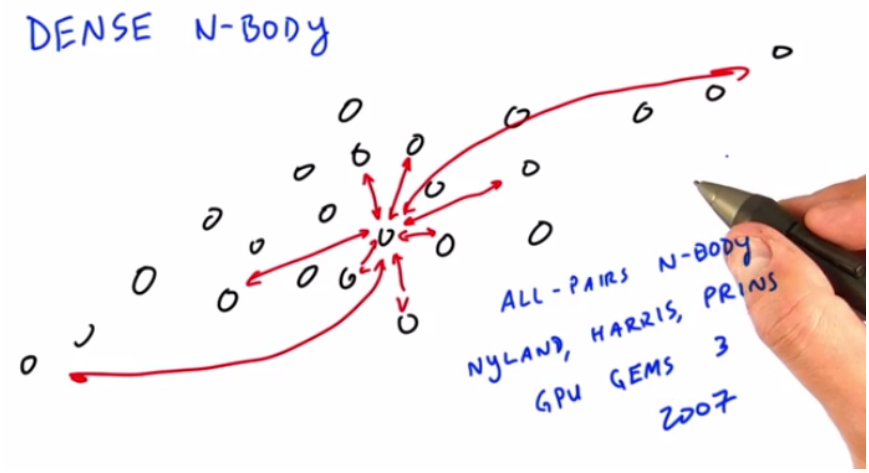
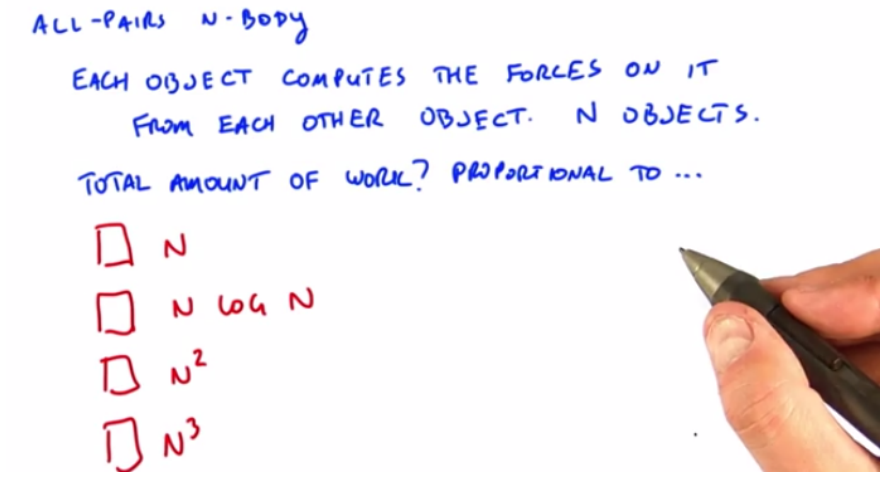
CS344 Introduction to Parallel Programming

Lesson 6.1: Parallel Computing Pattern Part A

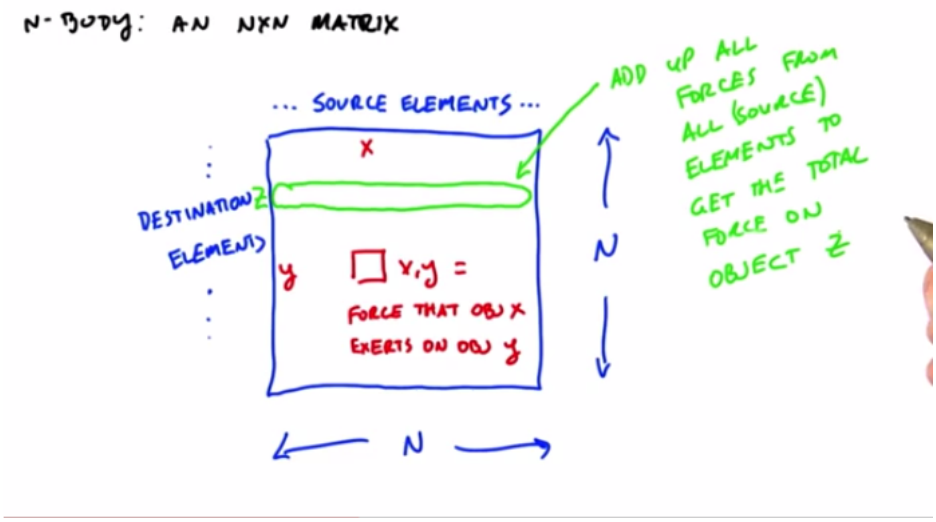
L6.1-6.2-Adding to Your GPU Computing Toolbox

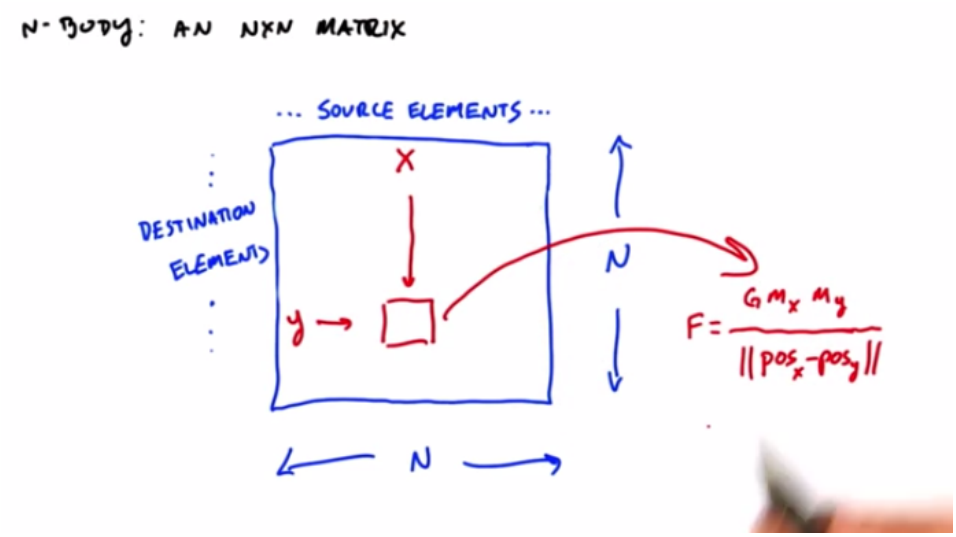
L6.1-6.3-All Pairs N-Body

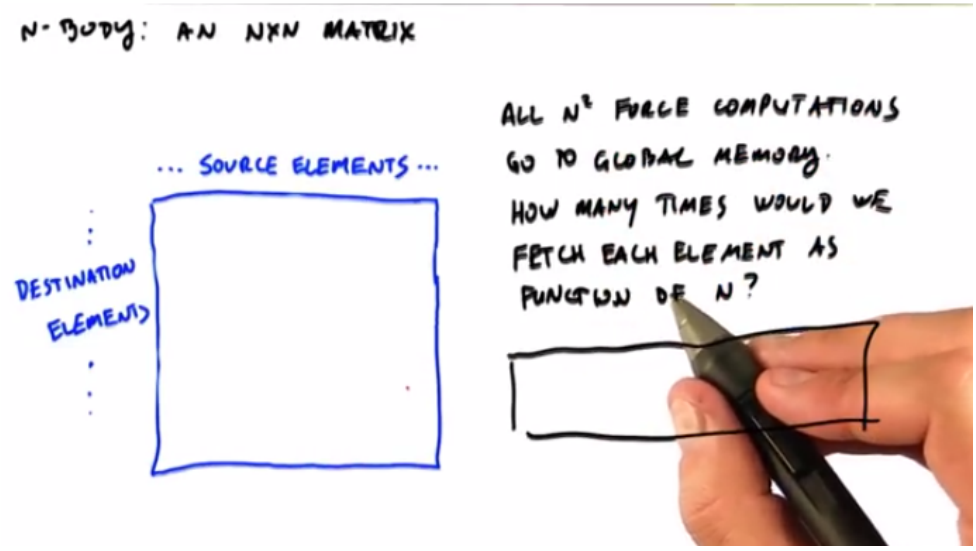




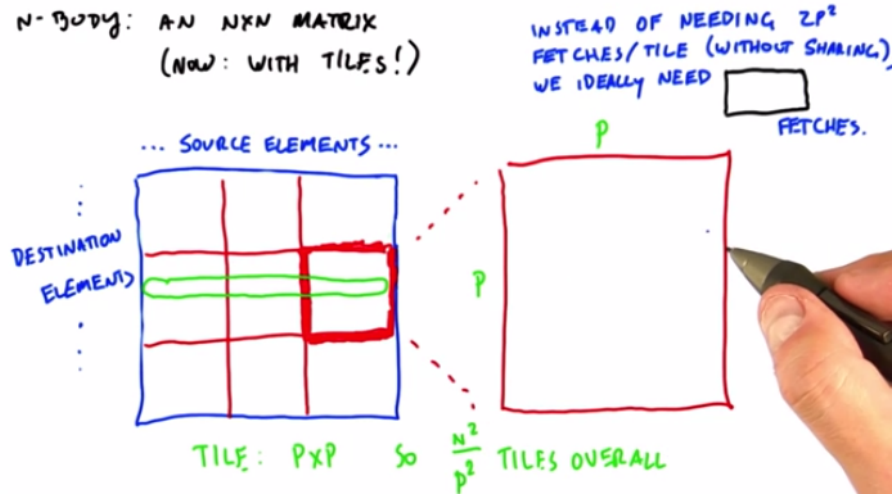
L6.1-6.4-How to Implement Dense N-Body as Simply As Possible



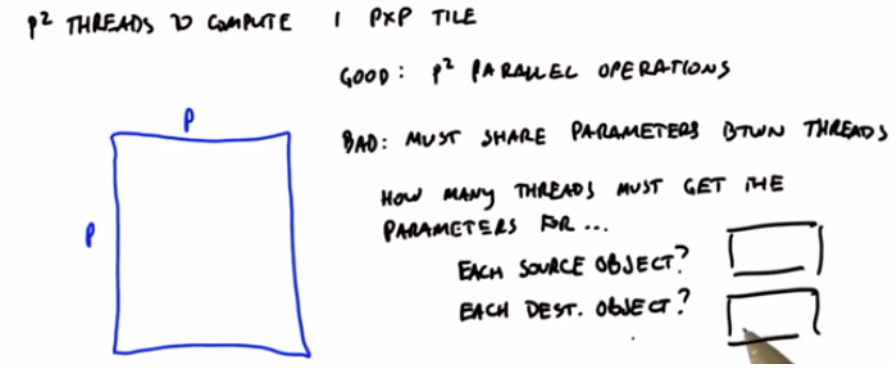




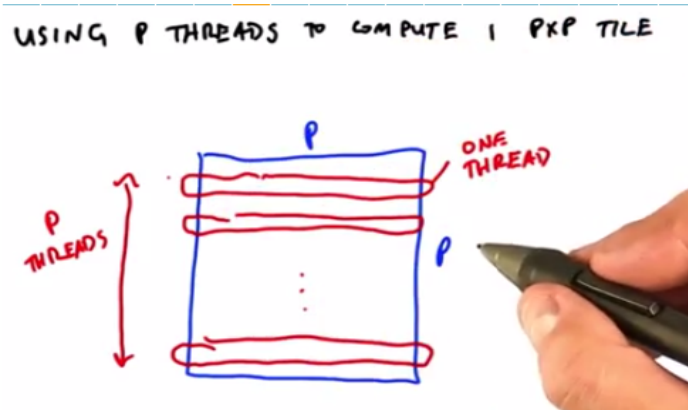
L6.1-6.5-Dividing N by N Metric into Tiles



L6.1-6.6-Downside to Using Tiles



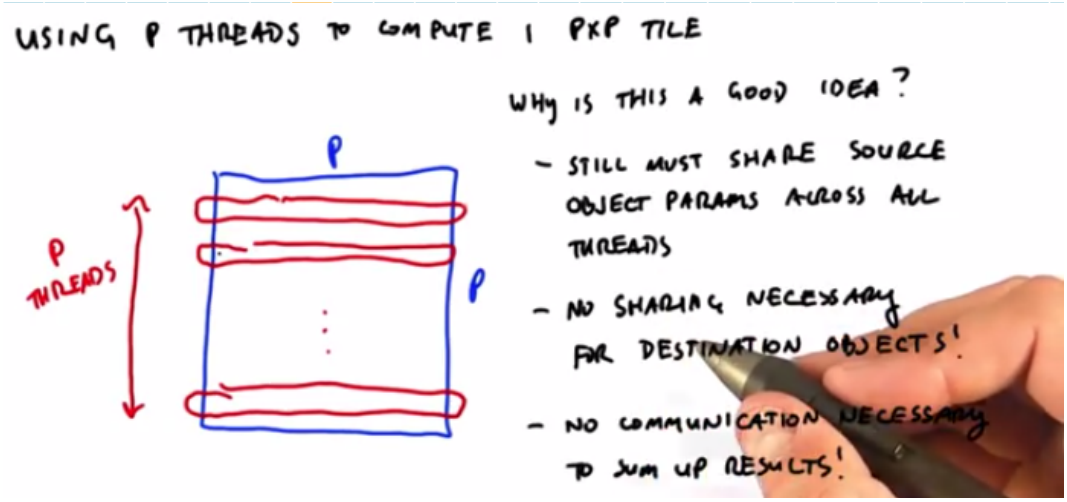
L6.1-6.7-Using on P Threads

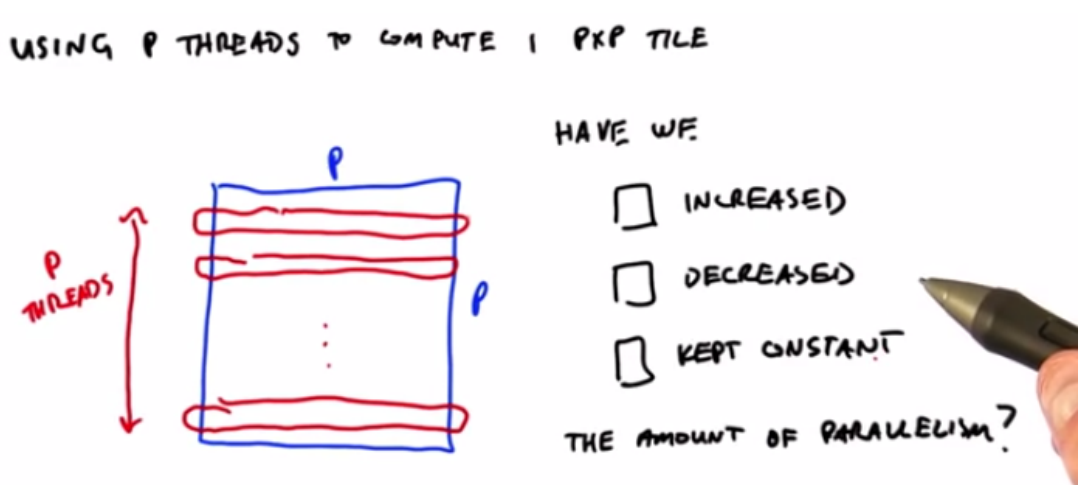



```
__device__ float3
tile_calculation(Params myParams, float3 force) {
  int i;
  extern __shared__ Params[] sourceParams;
  for (i = 0; i < blockDim.x; i++) {
    force += bodyBodyInteraction(myParams, sourceParams[i]);
  }
  return force;
}
```

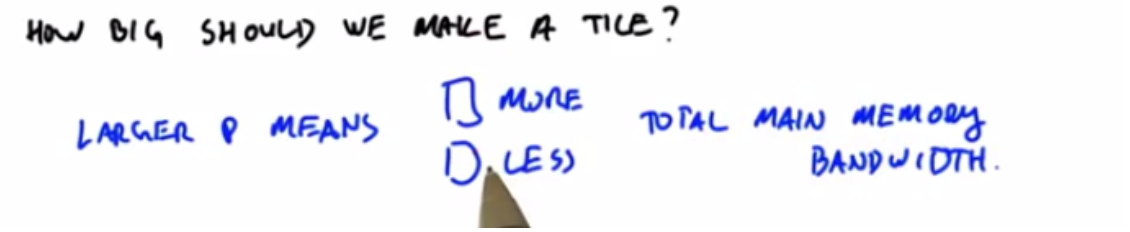
[CS344 Intro to Parallel Programming, Lesson 6.1]

L6.1-6.8-Why is This a Good Idea



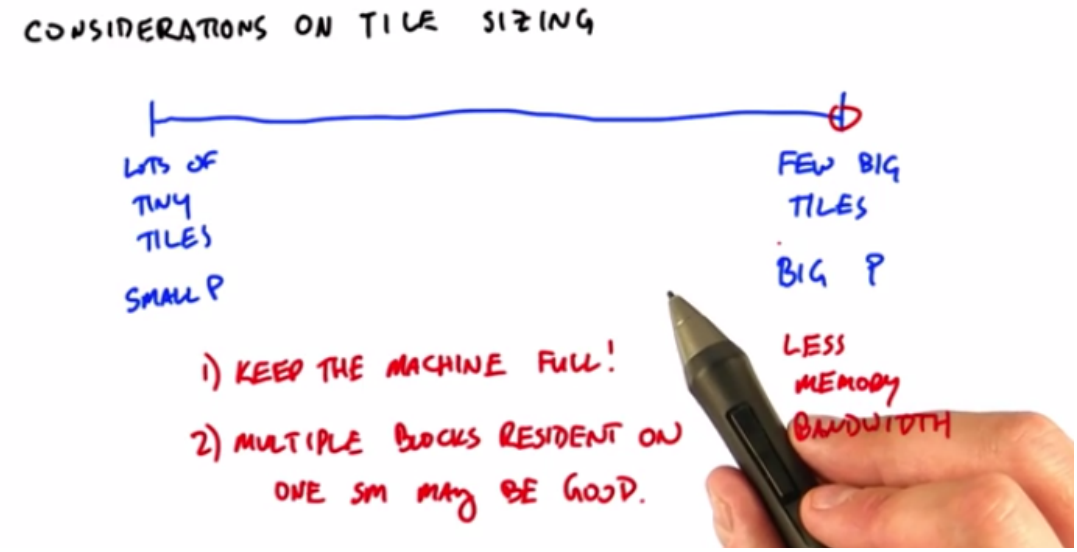


L6.1-6.9-How Big Should We Make a Tile



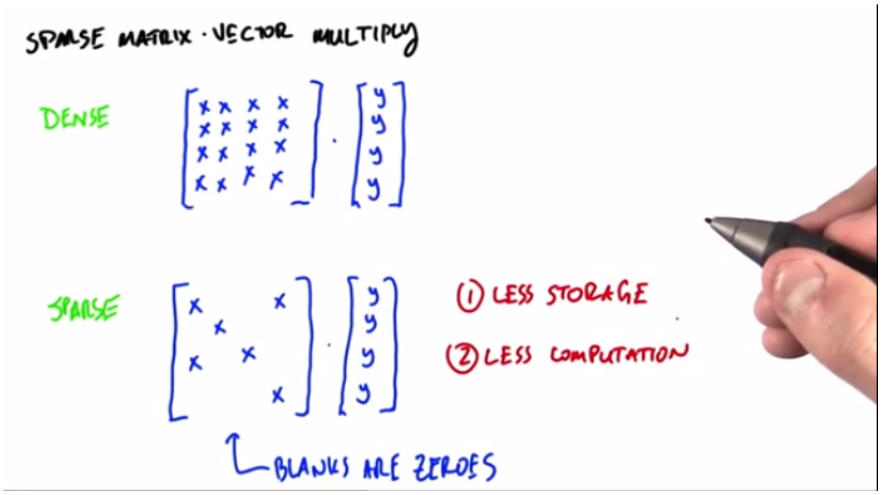
L6.1-6.10-What If P is Too Big or Small

L6.1-6.11-Determining the Size of the Tiles

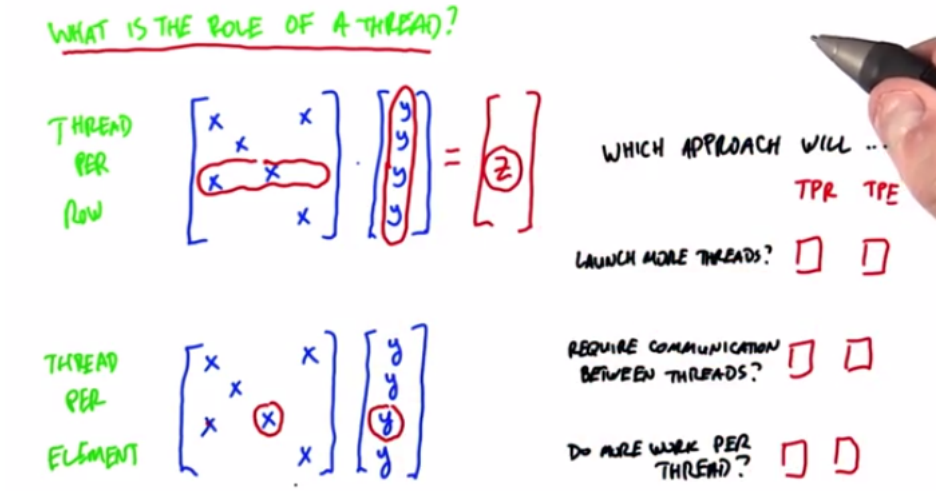


L6.1-6.12-All Pairs N-Body Summary

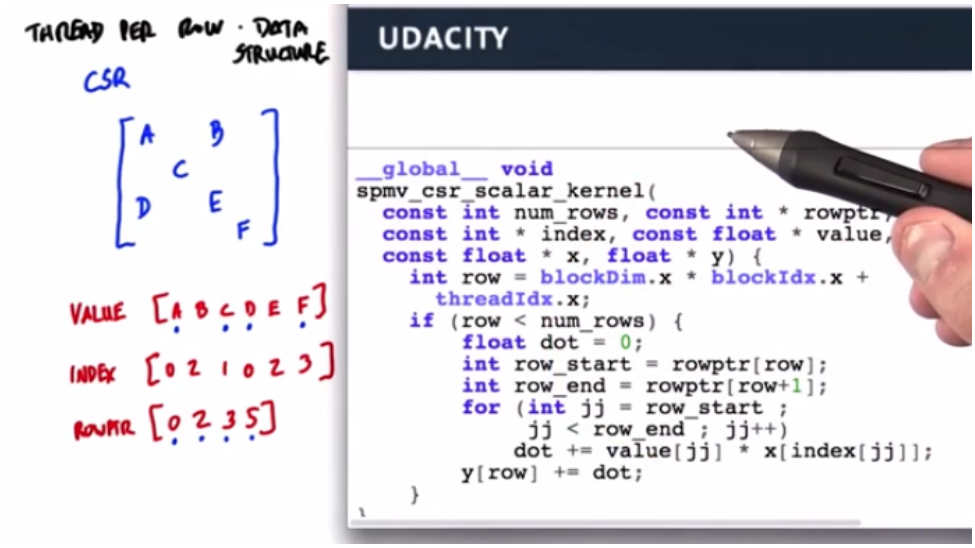
L6.1-6.13-SpMV



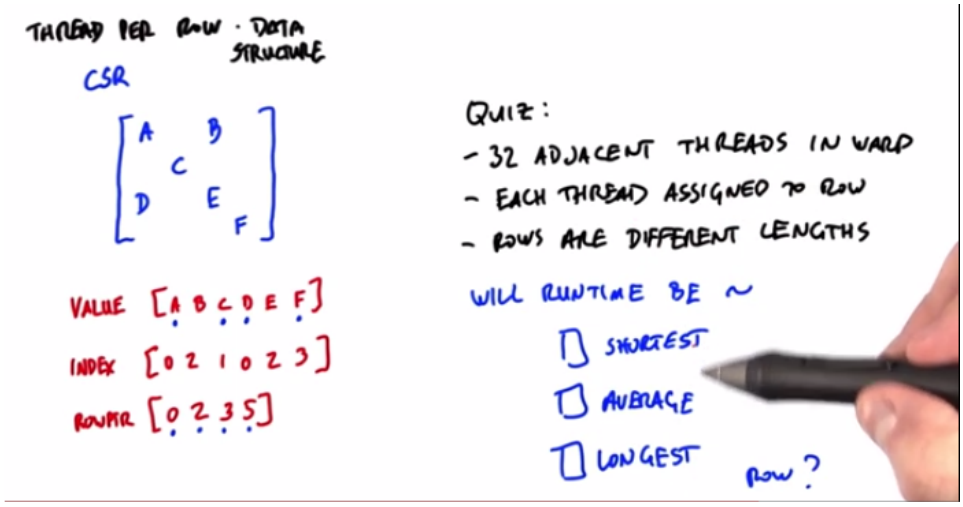
L6.1-6.14-Role of Thread in SpMv



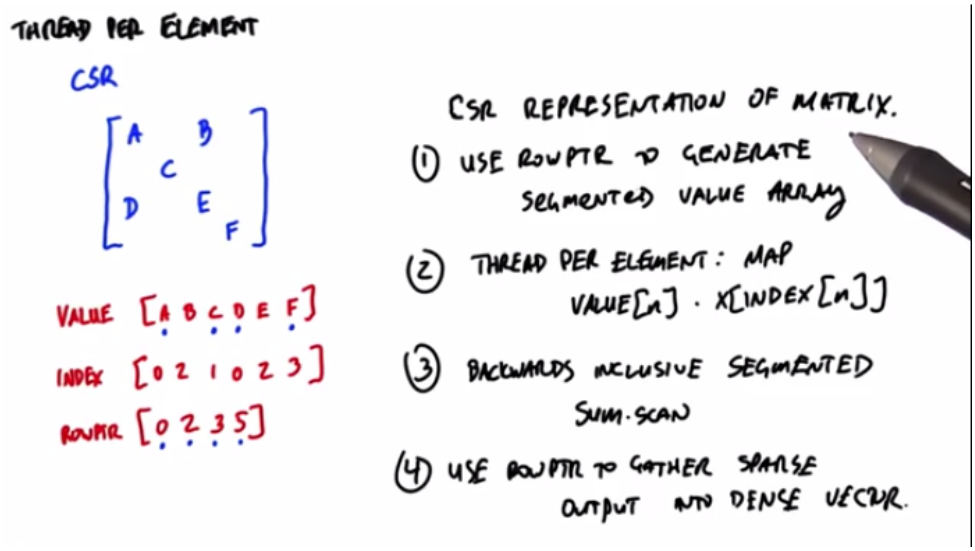
L6.1-6.15-Thread Per Row



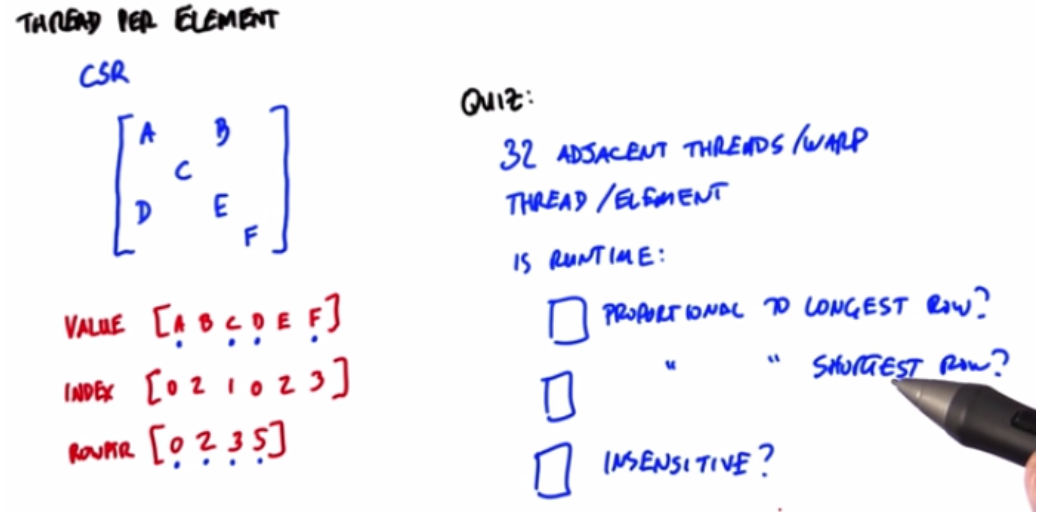
L6.1-6.16-Performance Analysis:TPR



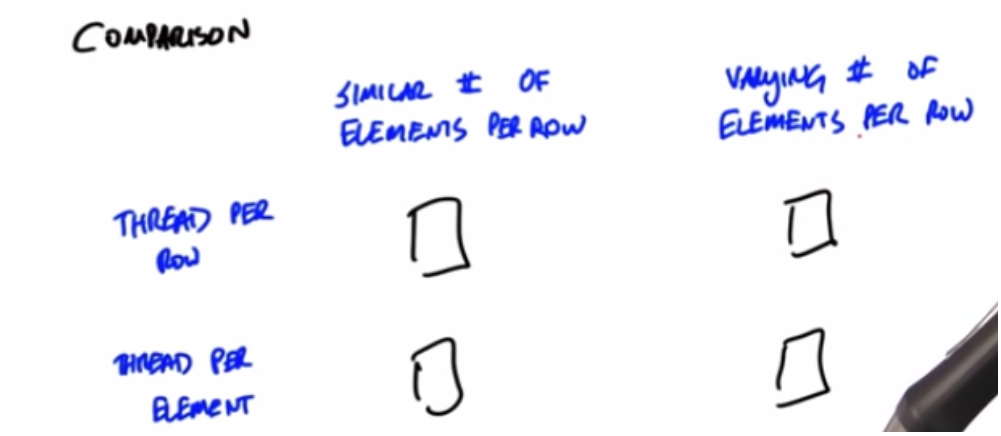
L6.1-6.17-Thread Per Element



L6.1-6.18-Performance Analysis:TPE

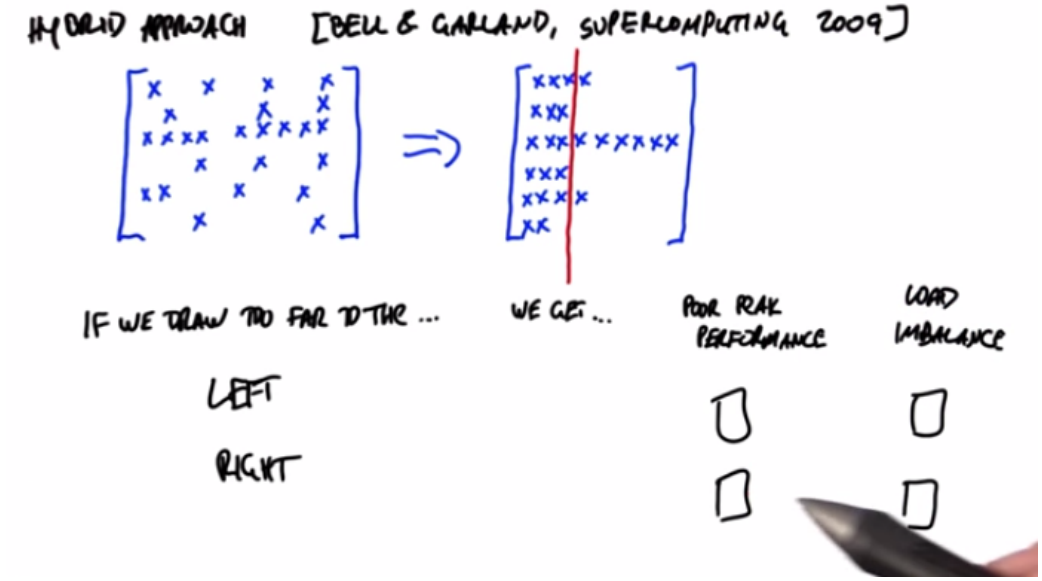


L6.1-6.19-Thread Per Row vs. Thread Per Element

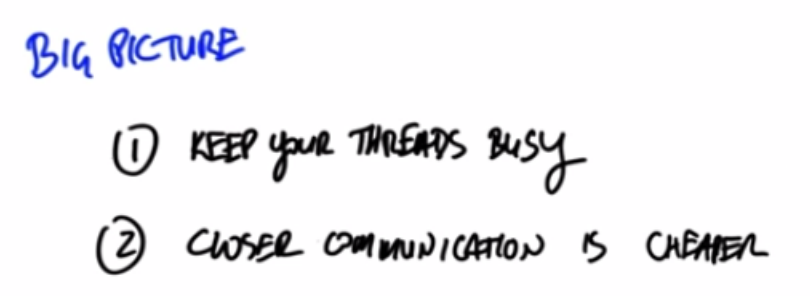


L6.1-6.20-Hybrid Approach

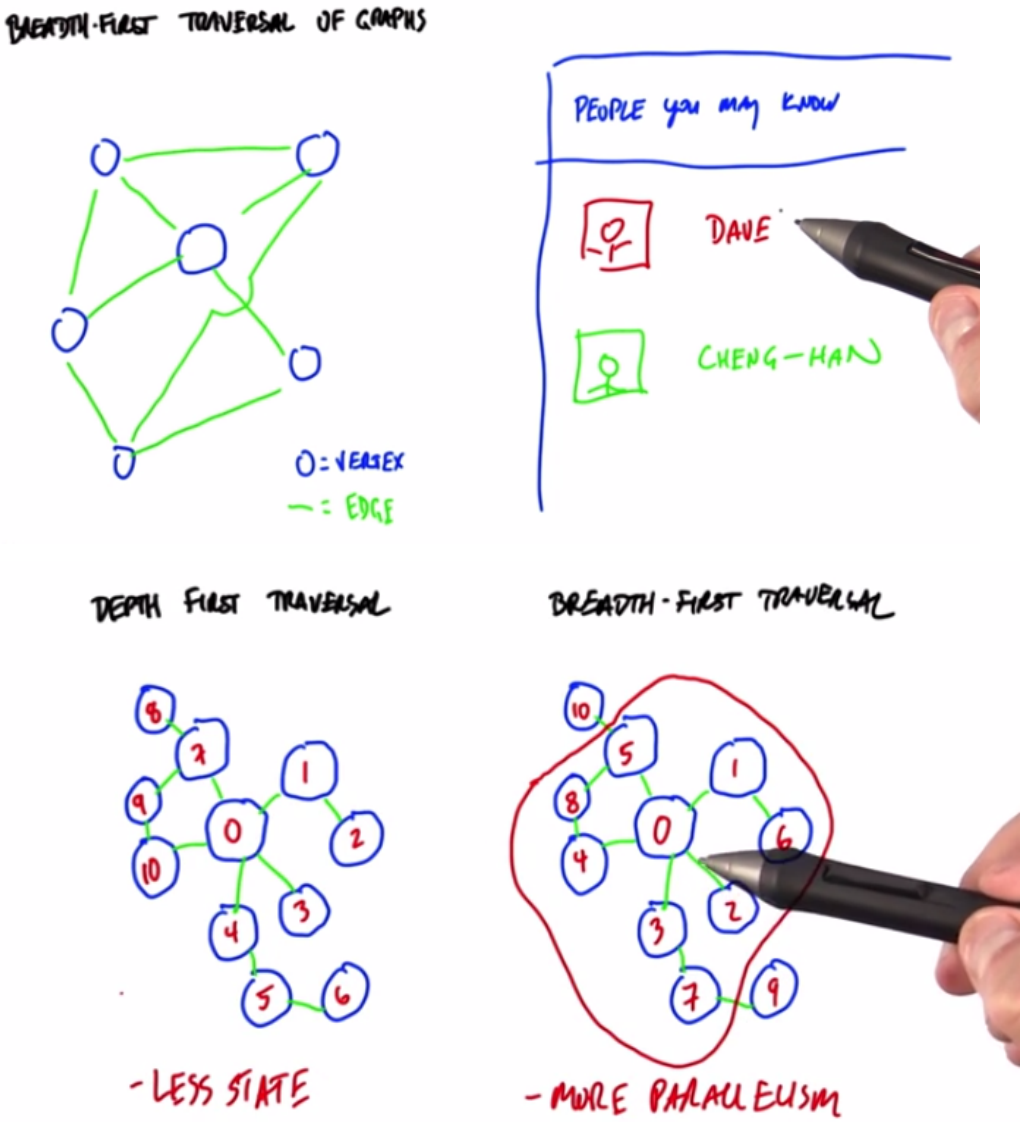
L6.1-6.21-A Quiz on the Hybrid Approach



L6.1-6.22-SpMV-The Big Picture

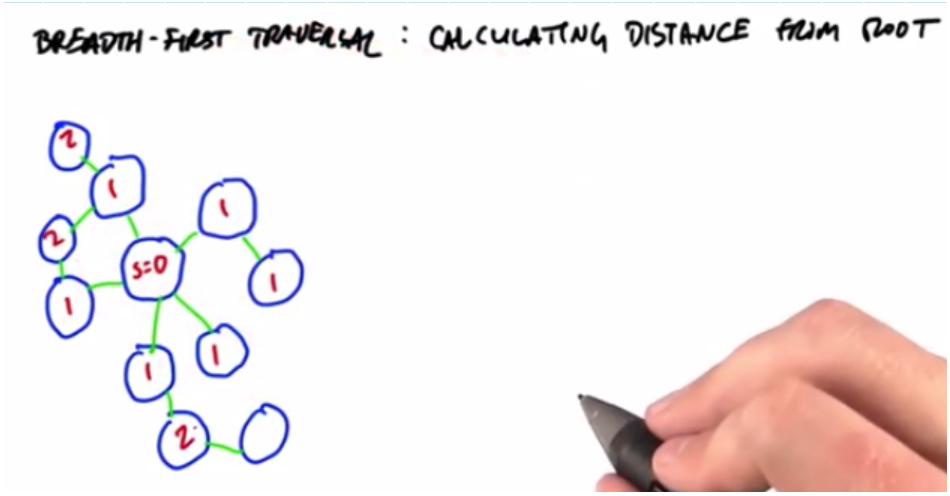


L6.1-6.23-Traversal of Graphs



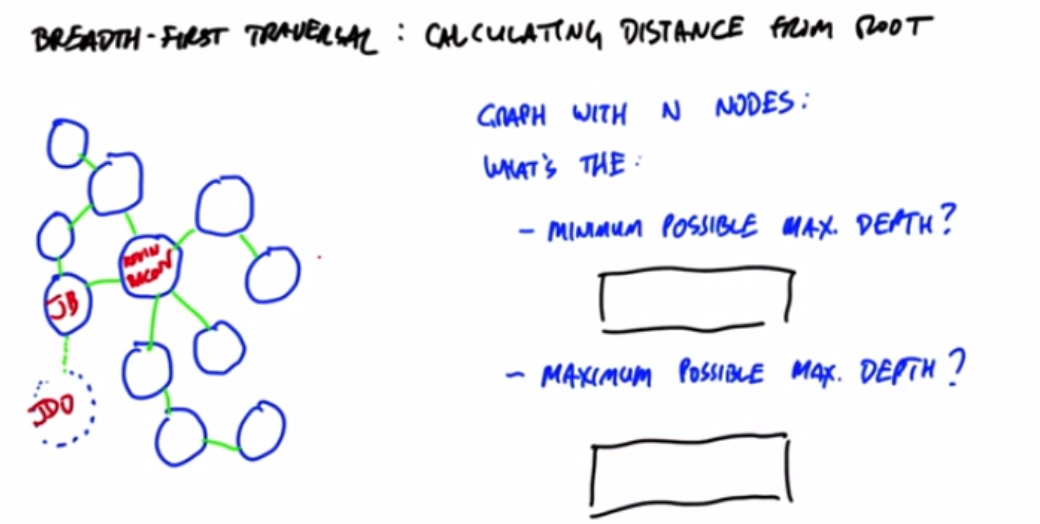
[Correction] On the right, 9 and 10 should be switched, and 9 should be included inside the red boundary.

L6.1-6.24-Calculating Distance From the Root

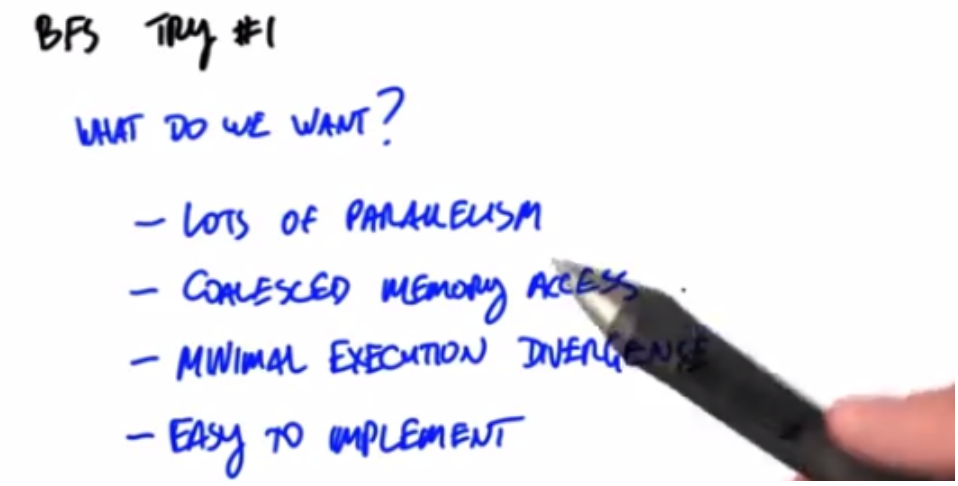


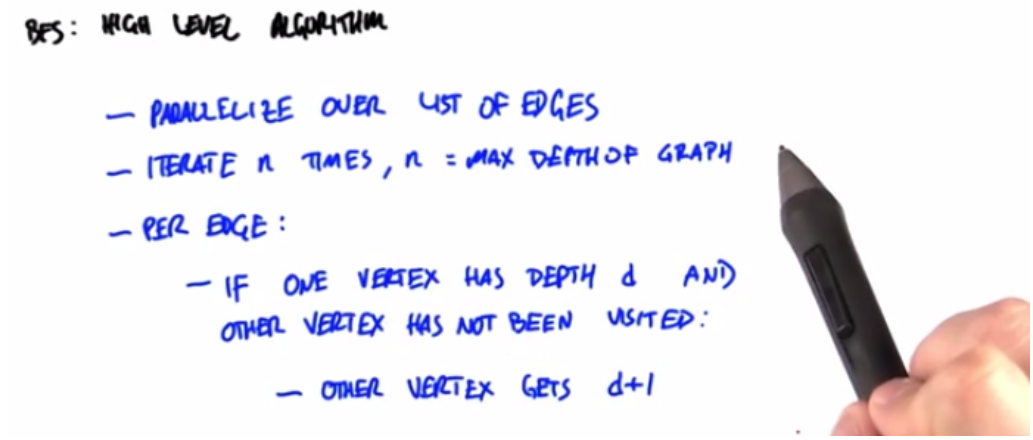
[Correction] far right 1 should be 2.

L6.1-6.25-Minimum and Maximum Possible Depth From Root

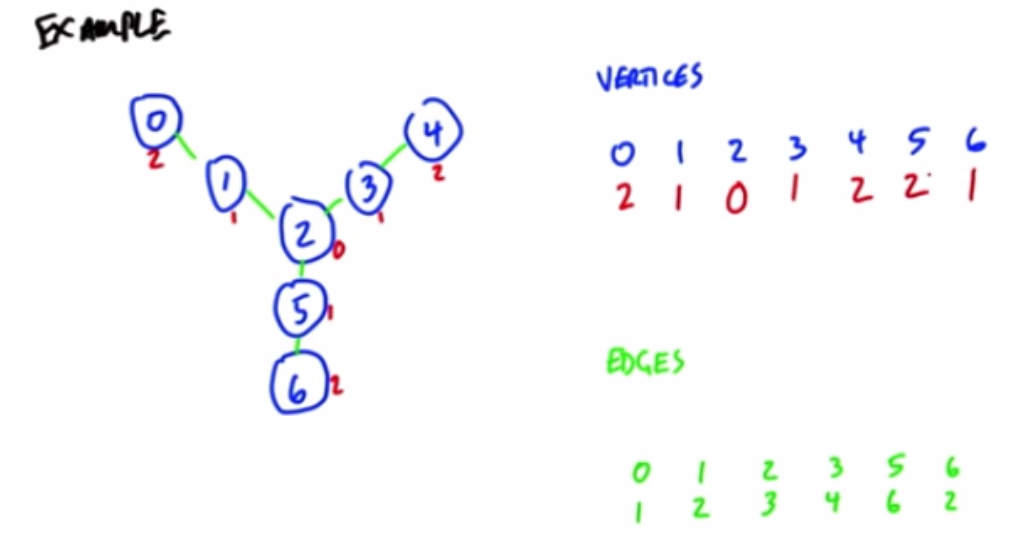


L6.1-6.26-BFS Try #1



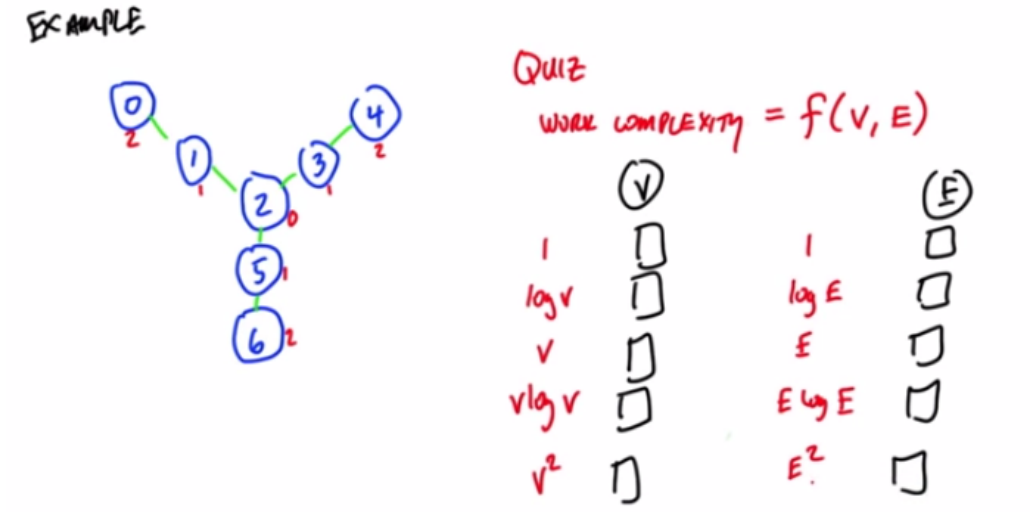


L6.1-6.27-BFS Try #1-An Example



[Correction]: Edges 012356/123425, vertices depth 2101212

L6.1-6.28-What is the Work Complexity



L6.1-6.29-BFS Code Part 1

```
// Initialization kernel: launch v threads
__global__ void
initialize_vertices( Vertex * vertices,
                     int starting_vertex,
                     int num_vertices )
{
    int v = blockDim.x * blockIdx.x + threadIdx.x;
    if (v == starting_vertex) vertices[v] = 0 else vertices[v] = -1;
}
```

L6.1-6.30-BFS Code Part 2

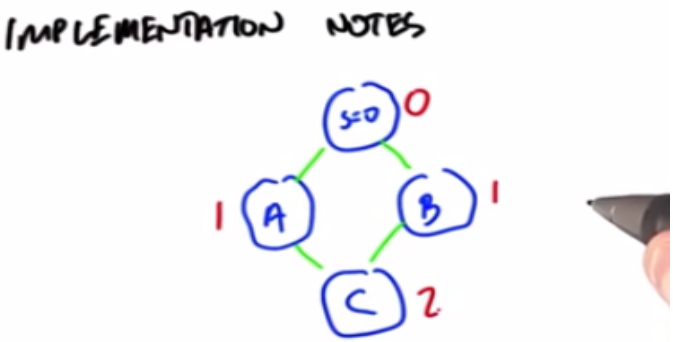
```
// BFS kernel: launch e threads
__global__ void
bfs( const Edge * edges,
     Vertex * vertices,
     int current_depth )
{
    int e = blockDim.x * blockIdx.x + threadIdx.x;
    int vfirst = edges[e].first;
    int dfirst = vertices[vfirst];
    int vsecond = edges[e].second;
    int dsecond = vertices[vsecond];
    if ((dfirst == current_depth)  && (dsecond == -1)) { // 1st visited, 2nd hasn't
      vertices[vsecond] = dfirst + 1;
    }
    if (                                              ) { // 2nd visited, 1st hasn't

    }
    // other two cases: {visited, visited}, {not, not} can be ignored
}
```

L6.1-6.31-BFS Code Part 3

```
// BFS kernel: launch e threads
__global__ void
bfs( const Edge * edges,
     Vertex * vertices,
     int current_depth )
{
    int e = blockDim.x * blockIdx.x + threadIdx.x;
    int vfirst = edges[e].first;
    int dfirst = vertices[vfirst];
    int vsecond = edges[e].second;
    int dsecond = vertices[vsecond];
    if ((dfirst == current_depth)  && (dsecond == -1)) { // 1st visited, 2nd hasn't
      vertices[vsecond] = dfirst + 1;
    }
    if ((dsecond == current_depth)  && (dfirst == -1)) { // 2nd visited, 1st hasn't
      vertices[vfirst] = dsecond + 1;
    }
    // other two cases: {visited, visited}, {not, not} can be ignored
}
```

L6.1-6.32-Do We Have to Worry About Race Condition



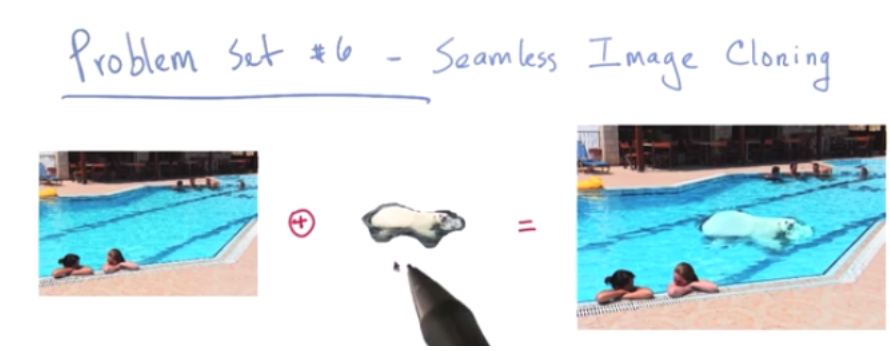
```
// snippet 1
while (!h_done) {
    bfs(edges, vertices)
    cudaMemcpy(&h_done, &d_done, sizeof(bool), cudaDeviceToHost);
}

// snippet 2
while (!h_done) {
    cudaMemcpy(&d_done, &h_true, sizeof(bool), cudaHostToDevice);
    bfs(edges, vertices)
    cudaMemcpy(&h_done, &d_done, sizeof(bool), cudaDeviceToHost);
}

// snippet 3
    if ((vfirst != -1)  && (vsecond == -1)) { // first visited, second hasn't
      vertices[vsecond] = vfirst + 1;
      done = false;
    }
    if ((vfirst == -1)  && (vsecond != -1)) { // second visited, first hasn't
      vertices[vfirst] = vsecond + 1;
      done = false;
    }
```

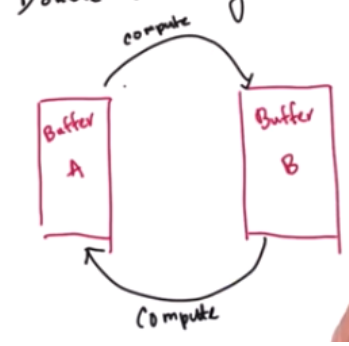
L6.1-6.33-BFS TRY #1 Summary

L6.1-6.34-Problem Set 6



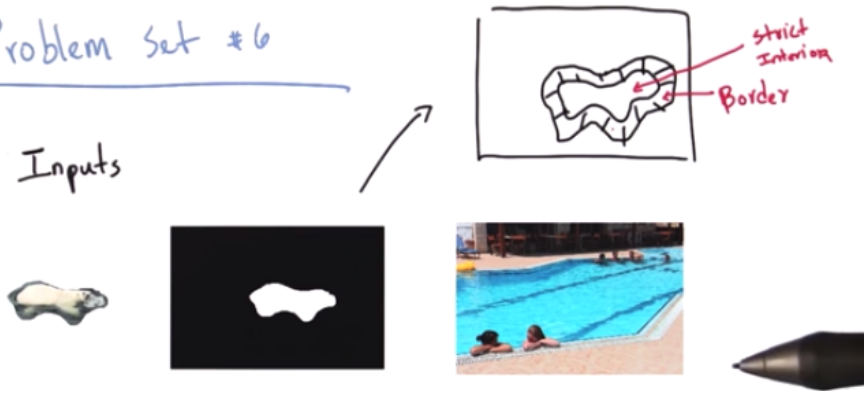
Problem Set #6

Double Buffering

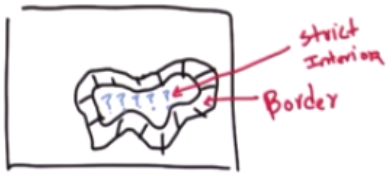


Problem Set #6

Inputs



Problem Set #6



$I_0 = 0$ | I_k |

Compute I_{k+1} as follows:

for i_{k+1} in I_{k+1},

$$i_{k+1} = \frac{A + B + C}{D}$$

A = sum of i_k's neighbors

B = sum of i's neighbors

C = difference between i and i's neighbors

D = number of i_k's neighbors