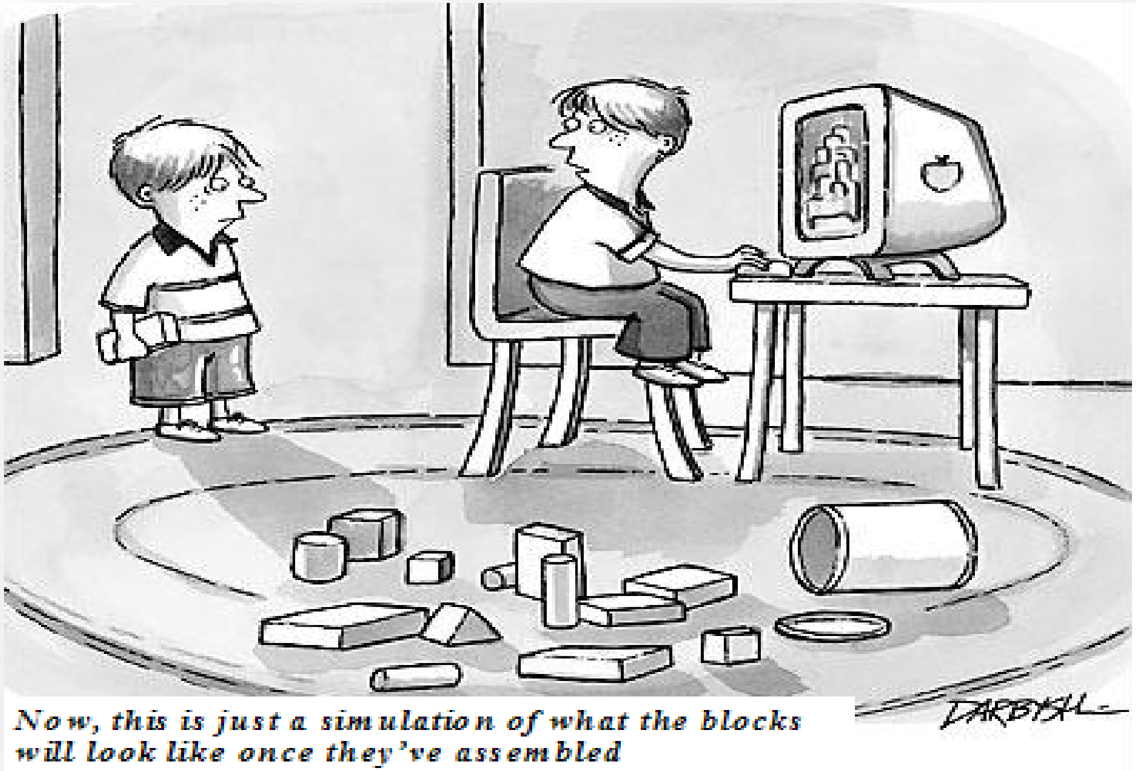
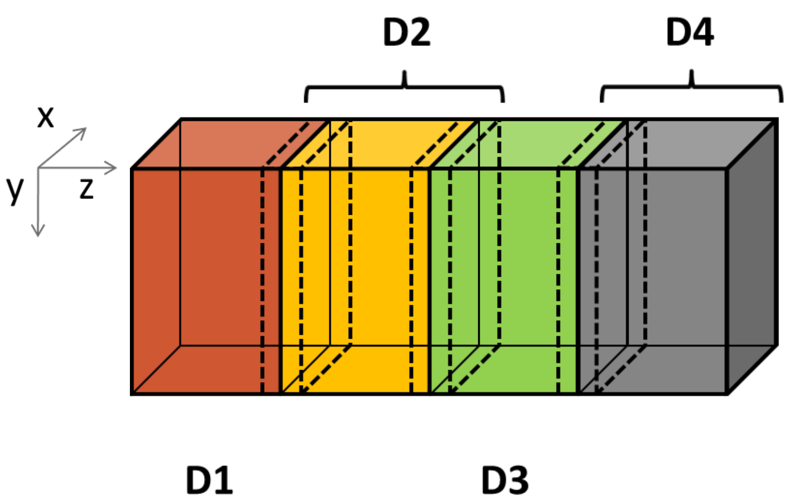
ECE569
Module 50

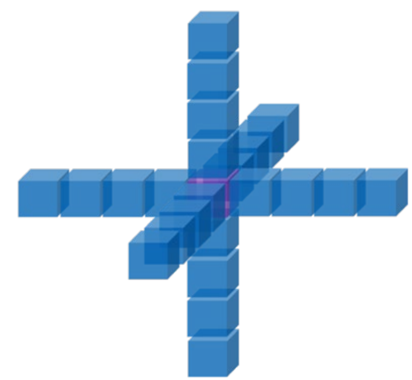


- MPI Example: 3D Heat Transfer

3D Heat Transfer

- Heat transfer based on Jacobi Iterative Method
  - in each iteration or time step, value of a point calculated as a weighted sum of 4 neighbors in each direction



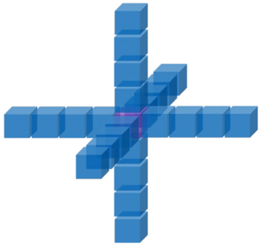


D

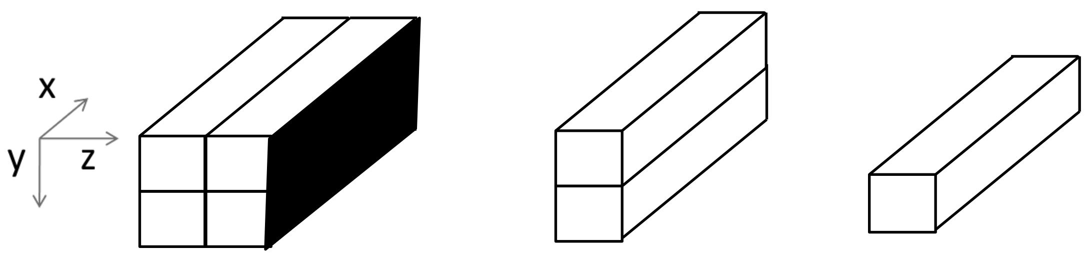
z=0 y=0		z=0 y=1		z=1 y=0		z=1 y=1		z=2 y=0		z=2 y=1		z=3 y=0		z=3 y=1	
x=0	x=1	x=0	x=1	x=0	x=1	x=0	x=1	x=0	x=1	x=0	x=1	x=0	x=1	x=0	x=1

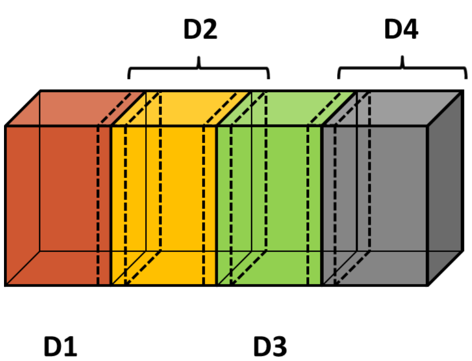
16 element partitioned into
4 elements (z direction)
2 elements in y ,
2 elements in x

Wave Propagation: Kernel Code



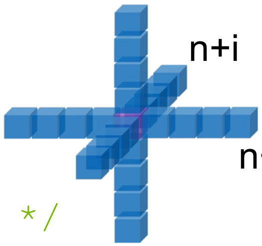
```
/* Coefficients used to calculate the laplacian */
__constant__ float coeff[5];

__global__ void wave_propagation(float *next, float *in,
                    float *prev, float *velocity, dim3 dim)
{
    unsigned x = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned y = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned z = threadIdx.z + blockIdx.z * blockDim.z;

    /* Point index in the input and output matrixes */
    unsigned n = z * dim.x * dim.y + y * dim.x + x ;

    /* Only compute for points within the matrixes */
    if(x < dim.x && y < dim.y && z < dim.z) {

        /* Calculate the contribution of each point to the laplacian */
        float laplacian = coeff[0] + in[n];
```

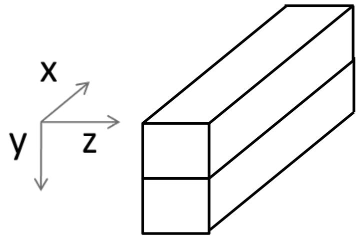


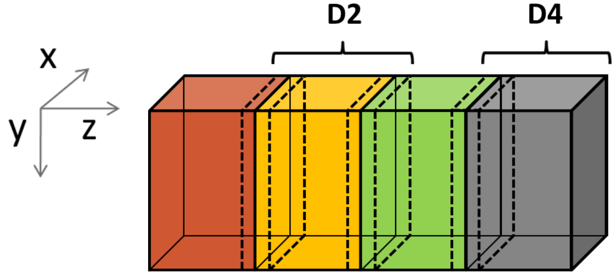


Wave Propagation: Time Integration

```
        for(int i = 1; i < 5; ++i) {
            laplacian += coeff[i] *
                          (in[n - i] + /* Left */
                          in[n + i] + /* Right */
                          in[n - i * dim.x] + /* Top */
                          in[n + i * dim.x] + /* Bottom */
                          in[n - i * dim.x * dim.y] + /* Behind */
                          in[n + i * dim.x * dim.y]);  /* Front */
        }

        /* Time integration */
        next[n] = velocity[n] * laplacian + 2 * in[n] - prev[n];
    }
}
```







Wave Propagation: Main Process

```
int main(int argc, char *argv[]) {
    int pad = 0, dimx  = 480+pad, dimy  = 480, dimz  = 400, nreps = 100;
    int pid=-1, np=-1;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    if(np < 3) {
        if(0 == pid) printf("Needed 3 or more processes.\n");
        MPI_Abort( MPI_COMM_WORLD, 1 ); return 1;
    }
    if(pid < np - 1)
        compute_node(dimx, dimy, dimz / (np - 1), nreps);
    else
        data_server( dimx,dimy,dimz, nreps );

    MPI_Finalize();
    return 0;
}
```

number of iterations that need to be done for all the data points in the grid.

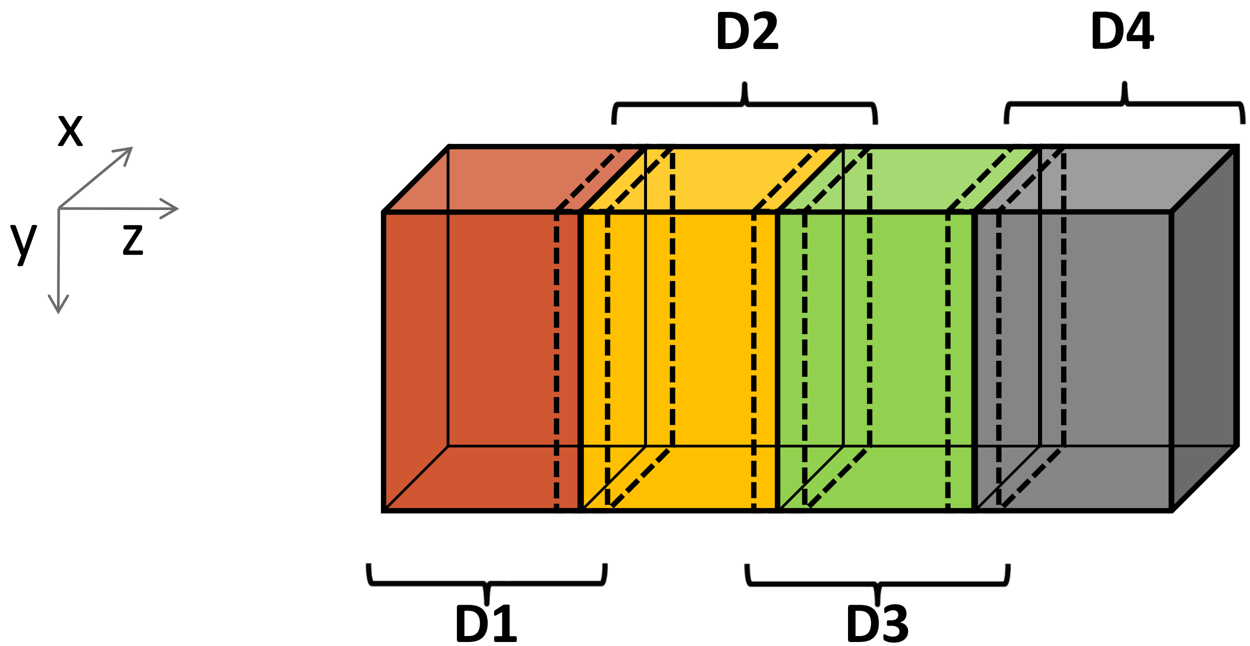
Stencil Code: Server Process (I)

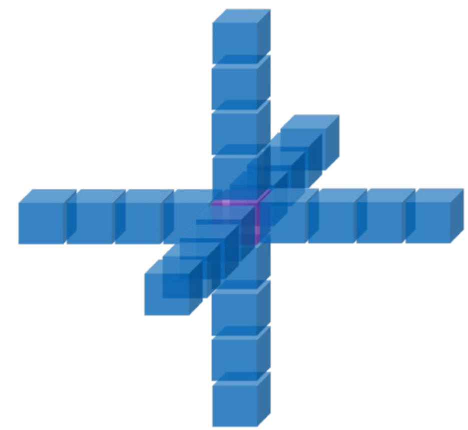
```
void data_server(int dimx, int dimy, int dimz, int nreps) {
    int np, num_comp_nodes = np – 1, first_node = 0, last_node = np - 2;
    unsigned int num_points = dimx * dimy * dimz;
    unsigned int num_bytes  = num_points * sizeof(float);
    float *input=0, *output = NULL, *velocity = NULL;

     /* Set MPI Communication Size */
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    /* Allocate input data */
    input = (float *)malloc(num_bytes);
    output = (float *)malloc(num_bytes);
    velocity = (float *)malloc(num_bytes);

      if(input == NULL || output == NULL || velocity == NULL) {
        printf("Server couldn't allocate memory\n");
        MPI_Abort( MPI_COMM_WORLD, 1 );
    }
    /* Initialize input data and velocity */
    random_data(input, dimx, dimy ,dimz , 1, 10);
    random_data(velocity, dimx, dimy ,dimz , 1, 10);
  float *send_address = input;
```

Stencil Code: Server Process (II)

```
/* Calculate number of shared points */
int edge_num_points = dimx * dimy * (dimz / num_comp_nodes + 4);
int int_num_points  = dimx * dimy * (dimz / num_comp_nodes + 8);
```

?

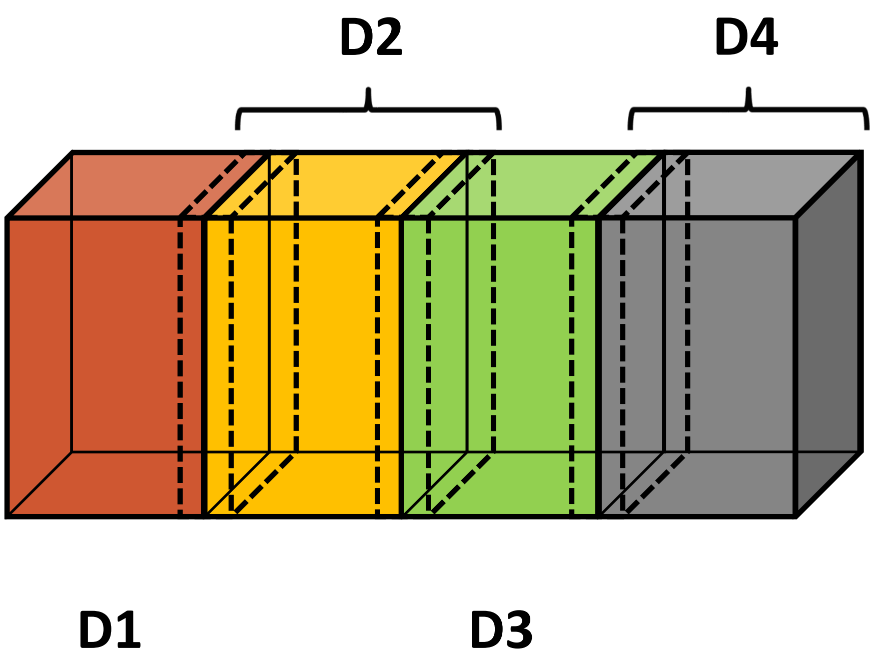




Internal partitions receive data from left and right

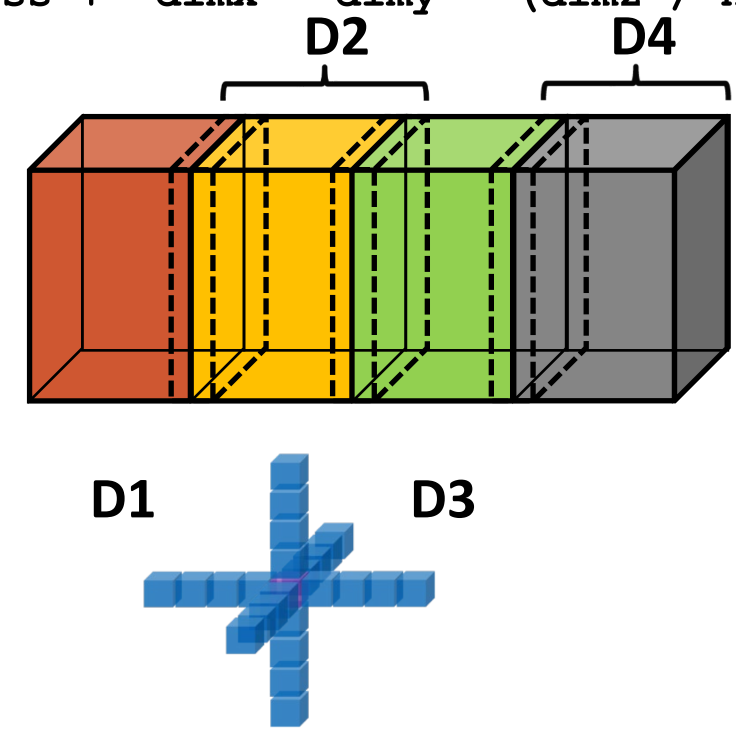
Stencil Code: Server Process (III)

```
/* Send input data to the first compute node */
MPI_Send(??????, ??????, MPI_REAL, ???????,
         DATA_DISTRIBUTE, MPI_COMM_WORLD );
```



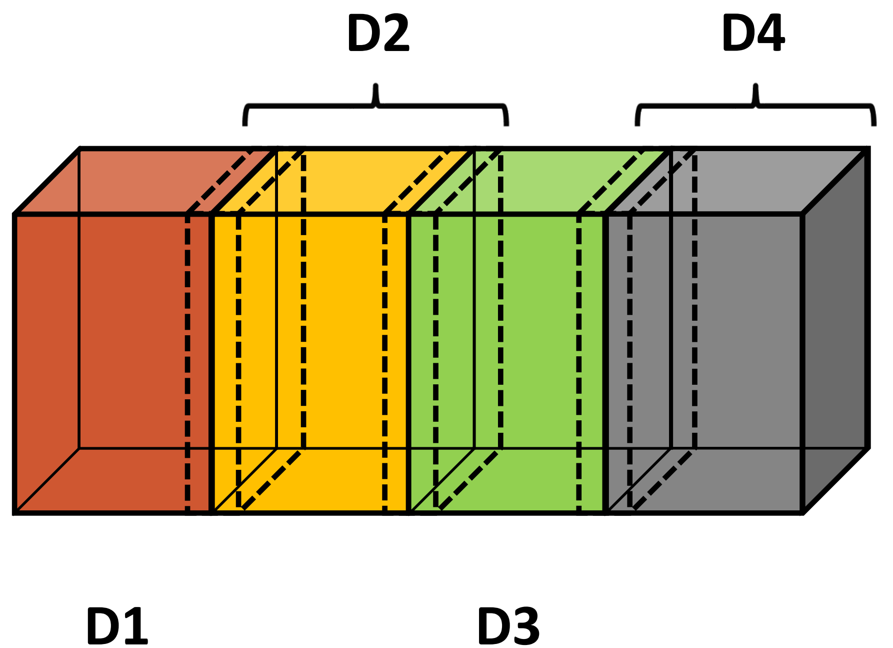
Stencil Code: Server Process (IV)

```
    /* adjust send address for internal nodes */
    send_address =____________________________________;

    /* Send input data to "internal" compute nodes */
    for(int process = 1; process < last_node; process++) {
        MPI_Send(send_address, int_num_points, MPI_FLOAT, process,
                 DATA_DISTRIBUTE, MPI_COMM_WORLD);
        send_address += dimx * dimy * (dimz / num_comp_nodes);
    }
```



Stencil Code: Server Process (V)

```
/* Send input data to the last compute node */
MPI_Send(send_address, edge_num_points, MPI_REAL, last_node,
        DATA_DISTRIBUTE, MPI_COMM_WORLD);
```



Stencil Code: Server Process (VI)

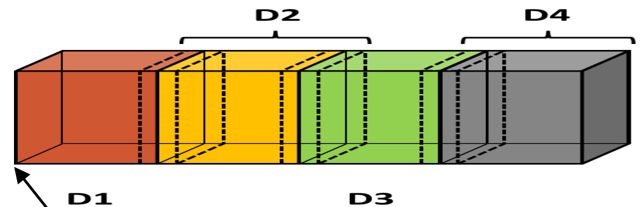
```
float *velocity_send_address = velocity;

/* Send velocity data to compute nodes */
for(int process = 0; process < last_node + 1; process++) {
        MPI_Send(velocity_send_address, edge_num_points,
          MPI_FLOAT, process, DATA_DISTRIBUTE, MPI_COMM_WORLD);
        velocity_send_address += dimx * dimy *
                                    (dimz / num_comp_nodes);
}

/* Wait for nodes to compute */
MPI_Barrier(MPI_COMM_WORLD);

/* Collect output data */
MPI_Status status;
for(int process = 0; process < num_comp_nodes; process++)
    MPI_Recv(output + process * num_points / num_comp_nodes,
        num_points / num_comp_nodes, MPI_FLOAT, process,
        DATA_COLLECT, MPI_COMM_WORLD, &status );
}
```

Stencil Code: Server Process (VII)

```
        /* Store output data */
    store_output(output, dimx, dimy, dimz);

    /* Release resources */
    free(input);
    free(velocity);
    free(output);
}
```

Stencil Code: Compute Process (I)

```
void compute_node_stencil(int dimx, int dimy, int dimz, int nreps ){
    int np, pid;
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    unsigned int num_points       = dimx * dimy * (dimz + 8);
    unsigned int num_bytes        = num_points * sizeof(float);
    unsigned int num_ghost_points = 4 * dimx * dimy;
    unsigned int num_ghost_bytes  = num_ghost_points *
                                     sizeof(float);
   int server_process = np-1;
    /* Alloc host memory */
    float *h_input = (float *)malloc(num_bytes);
    /* Alloc device memory for input and output data */
    float *rcv_address = h_input + num_ghost_points * (0 == pid);
    MPI_Recv(rcv_address, num_points, MPI_FLOAT, server_process,
             MPI_ANY_TAG, MPI_COMM_WORLD, &status );
```

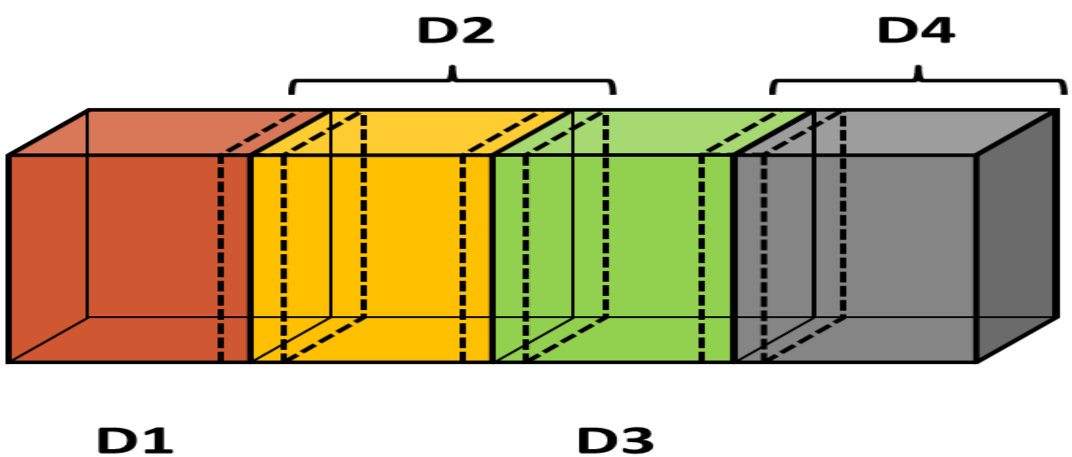


For simplicity both edge and internal nodes allocate same amount , +8 cells

Allocated +8 for node 0 receive buffer, left padding not used!

Stencil Code: Compute Process (II)

```
/* Alloc device memory for input and output data */
float *d_input = NULL;
cudaMalloc((void **)&d_input, num_bytes );
cudaMemcpy(d_input, h_input, num_bytes,
                                        cudaMemcpyHostToDevice);
 /* Compute time integration*/
 /* Transfer output back to host*/
 MPI_Barrier(MPI_COMM_WORLD);
 /* Send output data from compute node host CPU memory to
    the server node with MPI_Send() */
} /* end of compute node*/
```



MPI Sending and Receiving Data

- int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)
  - Sendbuf: Initial address of send buffer (choice)
  - Sendcount: Number of elements in send buffer (integer)
  - Sendtype: Type of elements in send buffer (handle)
  - Dest: Rank of destination (integer)
  - Sendtag: Send tag (integer)
  - Recvcount: Number of elements in receive buffer (integer)
  - Recvtype: Type of elements in receive buffer (handle)
  - Source: Rank of source (integer)
  - Recvtag: Receive tag (integer)
  - Comm: Communicator (handle)
  - Recvbuf: Initial address of receive buffer (choice)
  - Status: Status object (Status). This refers to the receive operation.

Stencil Code: Compute Process (III)

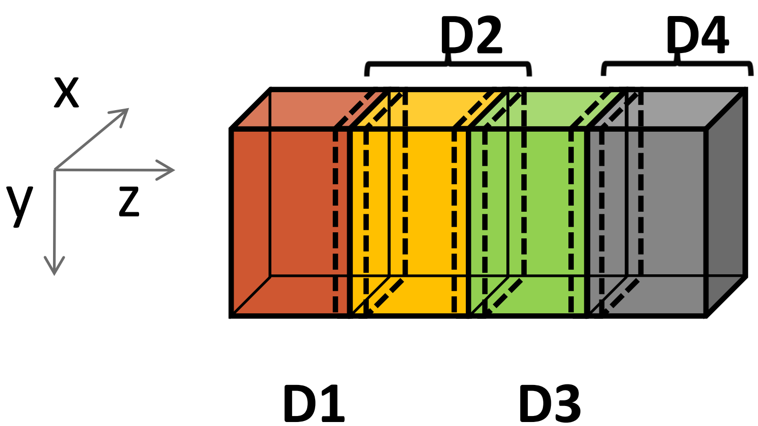
```
    /* Send data to left, get data from right */
    MPI_Sendrecv(h_left_boundary, num_halo_points, MPI_FLOAT,
                 left_neighbor,  i, h_right_halo,
                 num_halo_points, MPI_FLOAT, right_neighbor, i,
                 MPI_COMM_WORLD, &status );
    /* Send data to right, get data from left */
    MPI_Sendrecv(h_right_boundary, num_halo_points, MPI_FLOAT,
                 right_neighbor, i, h_left_halo,
                 num_halo_points, MPI_FLOAT, left_neighbor,  i,
                 MPI_COMM_WORLD, &status );

    cudaMemcpy(d_output+left_halo_offset,  h_left_halo,
                 num_halo_bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_output+right_ghost_offset, h_right_ghost,
                 num_halo_bytes, cudaMemcpyHostToDevice);
    cudaDeviceSynchronize();

    float *temp = d_output;
    d_output = d_input; d_input = temp;
}
```

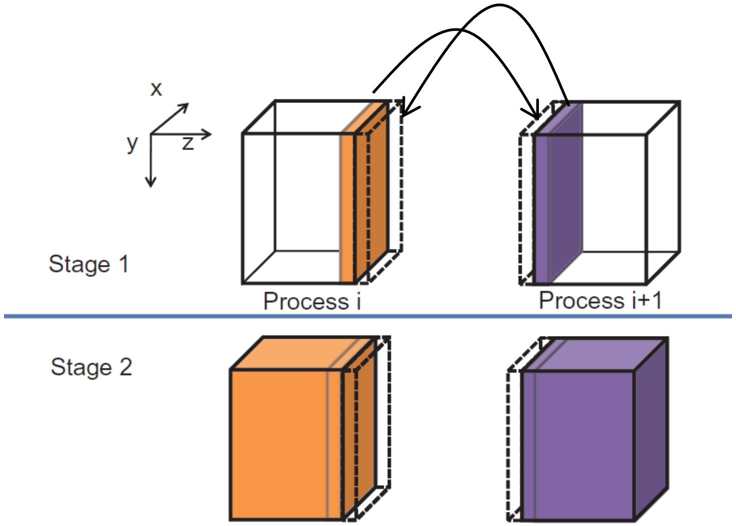
Stencil Code: Compute Process (IV)

```
/* Wait for previous communications */
MPI_Barrier(MPI_COMM_WORLD);

float *temp = d_output;
d_output = d_input;
d_input = temp;

/* Send the output, skipping halo points */
cudaMemcpy(h_output, d_output, num_bytes,
          cudaMemcpyDeviceToHost);
float *send_address = h_output + num_ghost_points;
MPI_Send(send_address, dimx * dimy * dimz, MPI_REAL,
      server_process, DATA_COLLECT, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);

/* Release resources */
free(h_input); free(h_output);
cudaFreeHost(h_left_ghost_own);
cudaFreeHost(h_right_ghost_own);
cudaFreeHost(h_left_ghost); cudaFreeHost(h_right_ghost);
cudaFree( d_input ); cudaFree( d_output );
}
```

Computation Efficiency

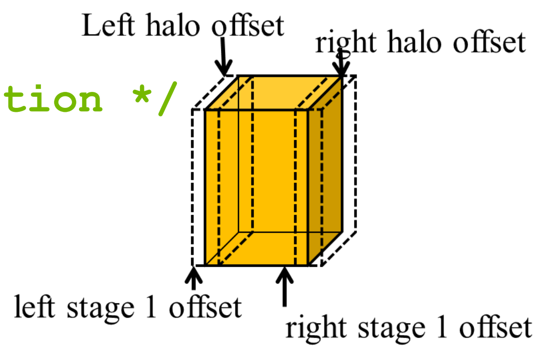
- Rather than having each node process all data, let each node first compute for its edge cells (Stage 1)
- Stream edge cell values to neighbors while processing rest of the data (Stage 2)





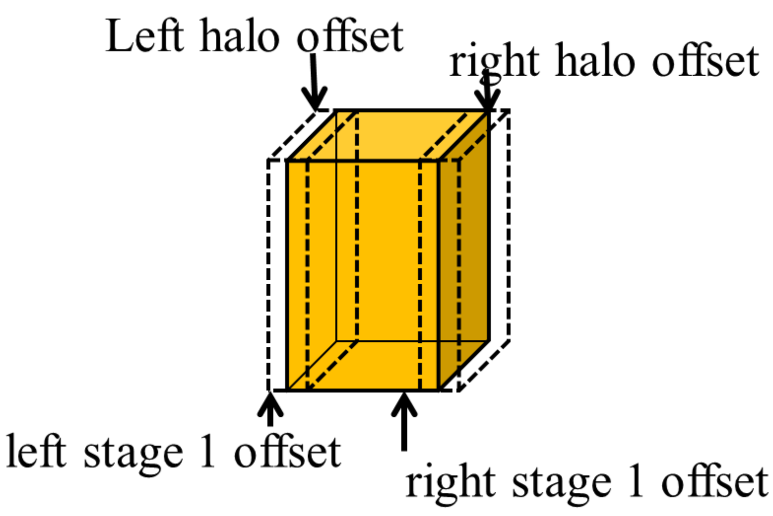
Compute Process Code

```
float *h_output = NULL, *d_output = NULL, *d_vsq = NULL;
float *h_output = (float *)malloc(num_bytes);
cudaMalloc((void **)&d_output, num_bytes );

float *h_left_boundary = NULL, *h_right_boundary = NULL;
float *h_left_halo = NULL, *h_right_halo = NULL;

/* Alloc host memory for ghost data */
cudaHostAlloc((void **)&h_left_boundary, num_ghost_bytes,
cudaHostAllocDefault);
cudaHostAlloc((void **)&h_right_boundary,num_ghost_bytes,
cudaHostAllocDefault);
cudaHostAlloc((void **)&h_left_halo,     num_ghost_bytes,
cudaHostAllocDefault);
cudaHostAlloc((void **)&h_right_halo,    num_ghost_bytes,
cudaHostAllocDefault);

/* Create streams used for stencil computation */
cudaStream_t stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);
```



Compute Process Code

```
MPI_Status status;
int left_neighbor  = (pid > 0)      ? (pid - 1) : MPI_PROC_NULL;
int right_neighbor = (pid < np - 2) ? (pid + 1) : MPI_PROC_NULL;

/* Upload stencil cofficients */
upload_coefficients(coeff, 5);

int left_halo_offset    = 0;
int right_halo_offset   = dimx * dimy * (4 + dimz);
int left_stage1_offset  = 0;
int right_stage1_offset = dimx * dimy * (dimz - 4);
int stage2_offset       = num_halo_points;

MPI_Barrier( MPI_COMM_WORLD );
 /* Compute boundary*/
/* Stage 1 */

/* Compute remaining points */
/* Stage 2 */
```



Compute Process Code

```
        /* Copy the data needed by other nodes to the host */
cudaMemcpyAsync(h_left_boundary, d_output + num_halo_points,
               num_halo_bytes, cudaMemcpyDeviceToHost, stream0 );

cudaMemcpyAsync(h_right_boundary,d_output + right_stage1_offset +
num_halo_points,num_halo_bytes, cudaMemcpyDeviceToHost, stream0 );

cudaStreamSynchronize(stream0);
```

Compute Process Code

```
    for(int i=0; i < nreps; i++) {
 /* Compute boundary values needed by other nodes first */
    launch_kernel(d_output + left_stage1_offset,
        d_input + left_stage1_offset, dimx, dimy, 12, stream0);
    launch_kernel(d_output + right_stage1_offset,
        d_input + right_stage1_offset, dimx, dimy, 12, stream0);

    /* Compute the remaining points */
    launch_kernel(d_output + stage2_offset, d_input + stage2_offset,
                dimx, dimy, dimz, stream1);
```

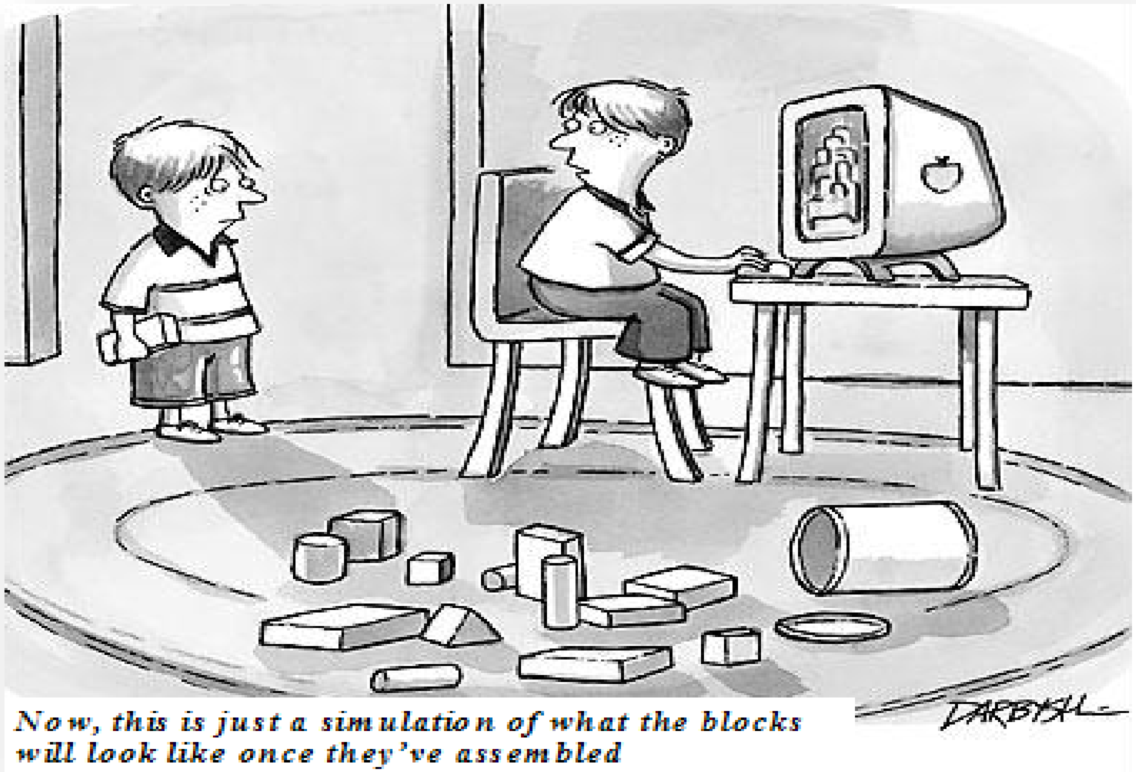
Compute Process Code (V)

```
    /* Send data to left, get data from right */
    MPI_Sendrecv(h_left_boundary, num_halo_points, MPI_FLOAT,
                 left_neighbor,  i, h_right_halo,
                 num_halo_points, MPI_FLOAT, right_neighbor, i,
                 MPI_COMM_WORLD, &status );
    /* Send data to right, get data from left */
    MPI_Sendrecv(h_right_boundary, num_halo_points, MPI_FLOAT,
                 right_neighbor, i, h_left_halo,
                 num_halo_points, MPI_FLOAT, left_neighbor,  i,
                 MPI_COMM_WORLD, &status );

    cudaMemcpyAsync(d_output+left_halo_offset,  h_left_halo,
                 num_halo_bytes, cudaMemcpyHostToDevice, stream0);
    cudaMemcpyAsync(d_output+right_ghost_offset, h_right_ghost,
                 num_halo_bytes, cudaMemcpyHostToDevice, stream0 );
    cudaDeviceSynchronize();

    float *temp = d_output;
    d_output = d_input; d_input = temp;
}
```

Next

- **Optimizing GPU Program**

ECE569
Module 51



- Optimizing GPU Programs

Fundamental Issues

- **Parallel computing requires that**
  - The problem can be decomposed into sub-problems that can be safely solved at the same time
  - The programmer structures the code and data to solve these sub-problems concurrently
- **The problems must be large enough to justify parallel computing and to exhibit exploitable concurrency.**

Optimizing GPU Programs

- ❑ Decrease arithmetic intensity
- ❑ Decrease time spent on memory operations
- ❑ Coalesce global memory accesses
- ❑ Do fewer memory operations per thread
- ❑ Avoid thread divergence
- ❑ Move all data to shared memory

Levels of Optimization

- Algorithm selection
- Basic efficiency principles
- Architecture specific optimizations
- Instruction level optimizations

Establishing a Strategy

- Analyze, Parallelize, Optimize, Deploy
  - Profile guided optimization
  - Deploy early
- Amdahl’s Law

Establishing an Upper Limit

- Theoretical Peak Bandwidth
  - Memory clock: 2GHz
  - Memory bus: 128 bits

  - 40-60% okay
  - 60-75% not bad
  - >75% very good

Establishing an Upper Limit

- Theoretical Peak Bandwidth
  - Memory clock: 1.6GHz
  - Memory bus: 128 bits
  - Kernel
    - Data size 1024x1024 elements (4 byte each)
    - 2 memory transactions per element
    - Execution time 0.5ms

DRAM Utilization (Global Memory)

- **Coalescing**
  – Utilization of the bytes delivered by memory

What can we do to improve bandwidth?

a. Increase the number of bytes delivered
b. Increase the latency (time between transactions)
c. Decrease the number of bytes delivered
d. Decrease the latency (time between transactions)

Data Transformation

```
struct my_Data {
 float a;
 float b;
 float c;
 float d;
};
struct my_Data data[128];
```

Array of Structures

```
int i  = threadIdx.x;
data[i].a++;
data[i].b += data[i].*
data[i].d;
```

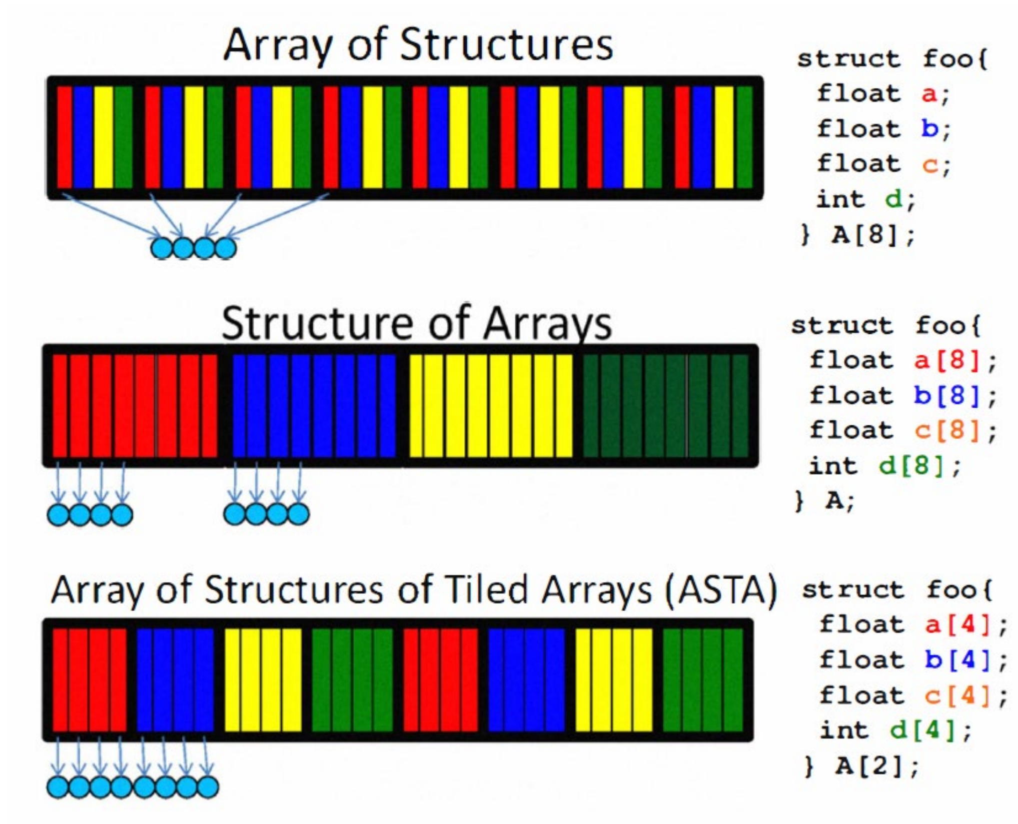
```
struct my_Data {
 float a[128];
 float b[128];
 float c[128];
 float d[128];
};
struct my_Data data;
```

Structure of arrays

```
int i  = threadIdx.x;
data.a[i]++;
data.b[i] += data.c[i]*
data.d[i];
```

In case your shared memory is performing worse

Data layout



. A. Stratton *et al.*, "Algorithm and Data Optimization Techniques for Scaling to Massively Threaded Systems," in *Computer*, vol. 45, no. 8, pp. 26-32, August 2012.

DRAM Utilization (Global Memory)

- **Coalescing**
  - Utilization of the bytes delivered by memory

- **Tiling**
  - Thread synchronization

What can we do to reduce the average time per thread?

a. Eliminate Syncthreads call
b. Reduce the number of threads per block
c. Increase the number of threads per block
d. Increase the number of blocks per SM

Tiling

- **If threads access overlapping parts of a data**
  – Buffering data into fast on-chip storage for repeated access
    - In CPU: resize data so that it fits into cache
      – Not so good for GPU with thousands of threads
      – Instead explicitly copy the data into the shared memory

Which one will benefit from tiling?

```
__global__ void avg(float* out,float* a,float* b,float* c,float*d, float*e){
  int i = threIdx.x;
  out[i] = (a[i]+b[i]+c[i]+d[i]+e[i])/5.0f;
}

__global__ void avg(float* out,float* a){
  int i = threIdx.x;
  out[i] = (a[i-2]+a[i-1]+a[i]+a[i+1]+a[i+2])/5.0f;
}
```

DRAM Utilization (Global Memory)

- **Coalescing**
  - Utilization of the bytes delivered by memory
- **Tiling**
  - Thread synchronization
- **Occupancy**
  - Thread blocks (16 /32)
  - Threads per block (1024)
  - Threads per SM (2048)
  - Registers per SM (64K)
  - Register per thread (255)
  - Shared memory per thread block(48KB/96KB/64KB)

Occupancy

- Given thread blocks per SM (16), threads per block (1024), Threads per SM (2048), Registers per SM (65,536 registers), Register per thread (255), Shared memory per thread block(48KB), calculate the maximum number of **thread blocks per SM** for a kernel assuming grid size is (32,32,1), block size of (32,32,1), 7 registers per thread, and 4KB shared memory per block. **Which resource prevents us from running more?**
  - Maximum number of threads/SM
  - Maximum number of registers/SM
  - Maximum shared memory/SM
  - Maximum thread blocks/SM

Thread divergence

- **Maximizing useful computations/second**
  – Minimize thread divergence
- **What is the maximum branch divergence penalty (slow down factor) for a CUDA thread block with 1024 threads?**

```
Switch(expression) {
 case1: …break;
 case2: …break;
 :
 :
 case32:…break;
}
```

Thread divergence

- **What will be the slowdown factor for the following expression in switch statement?**

```
switch (threadIdx.x%32) {
    case 0:
    case 1:
    :
    :
    case 31:
}
```

Assume kernel is launched as 1 block, 1024 threads per block.

Thread divergence

- **What will be the slowdown factor for the following expression in switch statement?**

```
switch (threadIdx.x%64) {
    case 0:
    case 1:
    :
    :
    case 63:
}
```

Assume kernel is launched as 1 block, 1024 threads per block.

Scatter-to-Gather transformation

- **Scatter:**
  - Threads are assigned to the inputs and each one is deciding where it needs to write
- **Gather:**
  - Threads are assigned to the output elements and each one is deciding where it needs to read from

Which one will run more efficiently? (i = threadIdx.x)

```
float val = in[i]/3.0f;
out[i-1] += val;
out[i] += val;
out[i+1] += val;
```

```
out[i]= (in[i-1] + in[i]
+ in[i+1])/3.0f
```

Privatization

- **Tiling**
- **Problem: what if threads write into the same address?**
  - Histogram
    - Local (per thread and thread blocks) and then global

Common optimization techniques

- **Suggested Reading:**
  - J. A. Stratton *et al*., "Algorithm and Data Optimization Techniques for Scaling to Massively Threaded Systems," in *Computer*, vol. 45, no. 8, pp. 26-32, August 2012.
  - J. A. Stratton *et al*., "Optimization and architecture effects on GPU computing workload performance," *2012 Innovative Parallel Computing (InPar)*, San Jose, CA, 2012, pp. 1-10.

Summary

- **Measure and improve memory bandwidth**
  - Assure sufficient occupancy
  - Coalesce global memory access
  - Minimize latency between accesses
    - Synchronization: too many threads waiting
  - Minimize thread divergence in a warp
  - Avoid code with if, switch
    - It is ok to have if statement for edge cases
  - Avoid workload imbalance across threads
  - Consider using built-in math functions ..sin();
  - Single precision vs. double precision
    - 3.14 vs 3.14f (single precision!)
  - Streams
    - Overlap computation and memory transfers

Easy to Learn

- **Takes time to master**