Programming Language Laboratory

Code Optimization Techniques for Graphics Processing Units

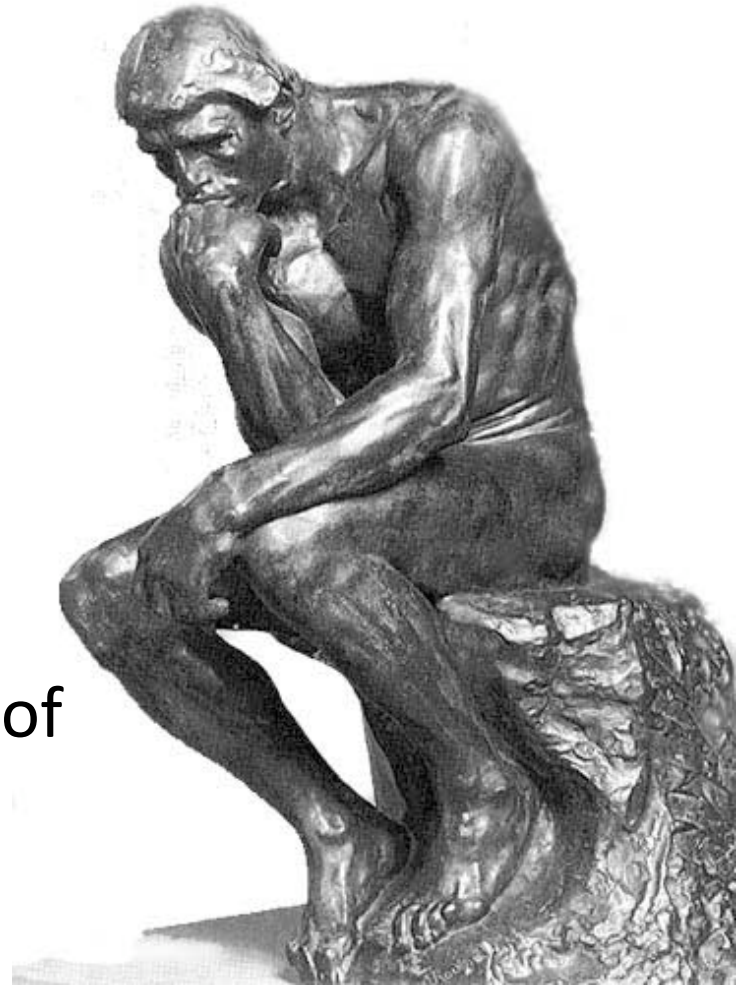
Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

fernando@dcc.ufmg.br

Memory Optimizations

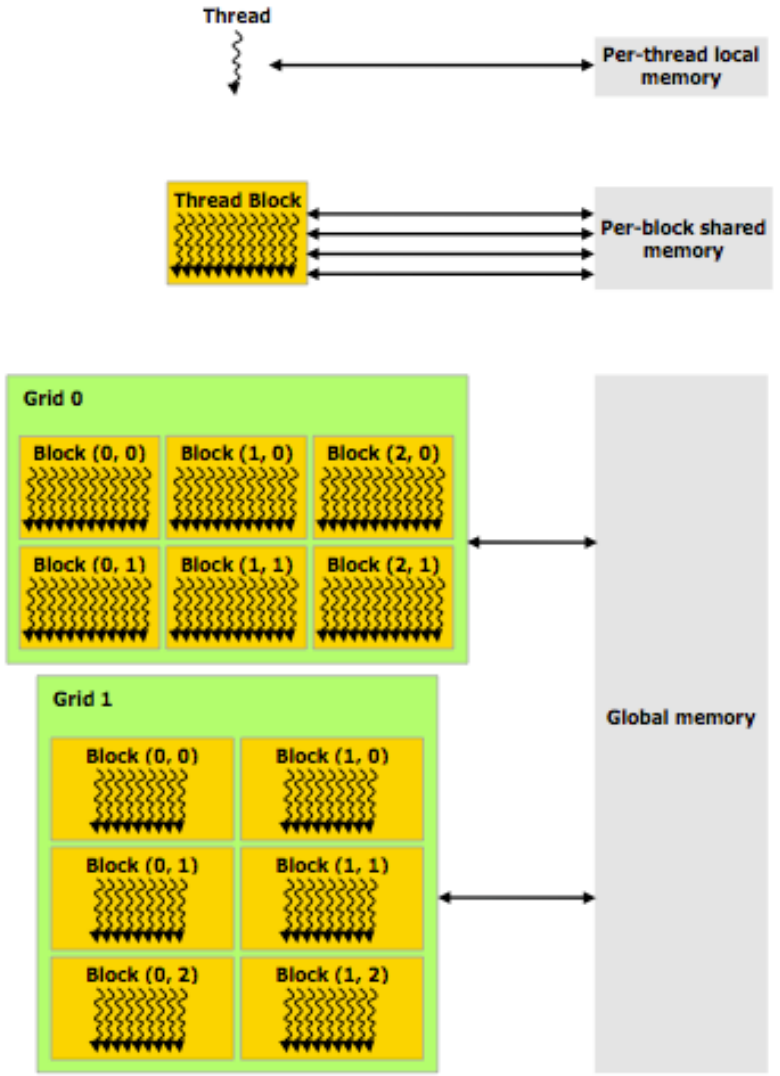
- When is it worthwhile running applications on the GPU?
- Which compiler optimizations do you know?
- Which optimizations make sense when running applications on a GPU?
- How to measure the performance of CUDA programs?



fernando@dcc.ufmg.br

Quick Review

- Each thread has local registers and local memory
- Threads are grouped in warps
  - Warps run in SIMD exec
- Warps are grouped in blocks
  - Shared memory + syncs
- Blocks are grouped in grids
  - Each grid represents a kernel



fernando@dcc.ufmg.br

Quick review

- How do different grids communicate?
- How do threads in the same block communicate?
- What is the effect of branches in the warp execution?
- What is the language that we use in CUDA?
  - How is this language different from traditional C?

fernando@dcc.ufmg.br

Latency versus Throughput

- What is latency?
- What is throughput?
- Which one is more important in a traditional CPU?
- Which one is more important on the GPU?

fernando@dcc.ufmg.br

Going to the Archives

- Do you remember the memory hierarchy that we find in traditional CPUs?
- Why do we have this hierarchy?
- GPUs are much more memory intensive than traditional CPUs. Why?
  - The GeForce 8800 processes **32 pixels** per clock. Each pixel contains a color (**3 bytes**) and a depth (**4 bytes**), which are **read** and **written**. On the average **16** extra bytes of information are read for each pixel. How many bytes are processed per clock?

fernando@dcc.ufmg.br

The GPU Archive

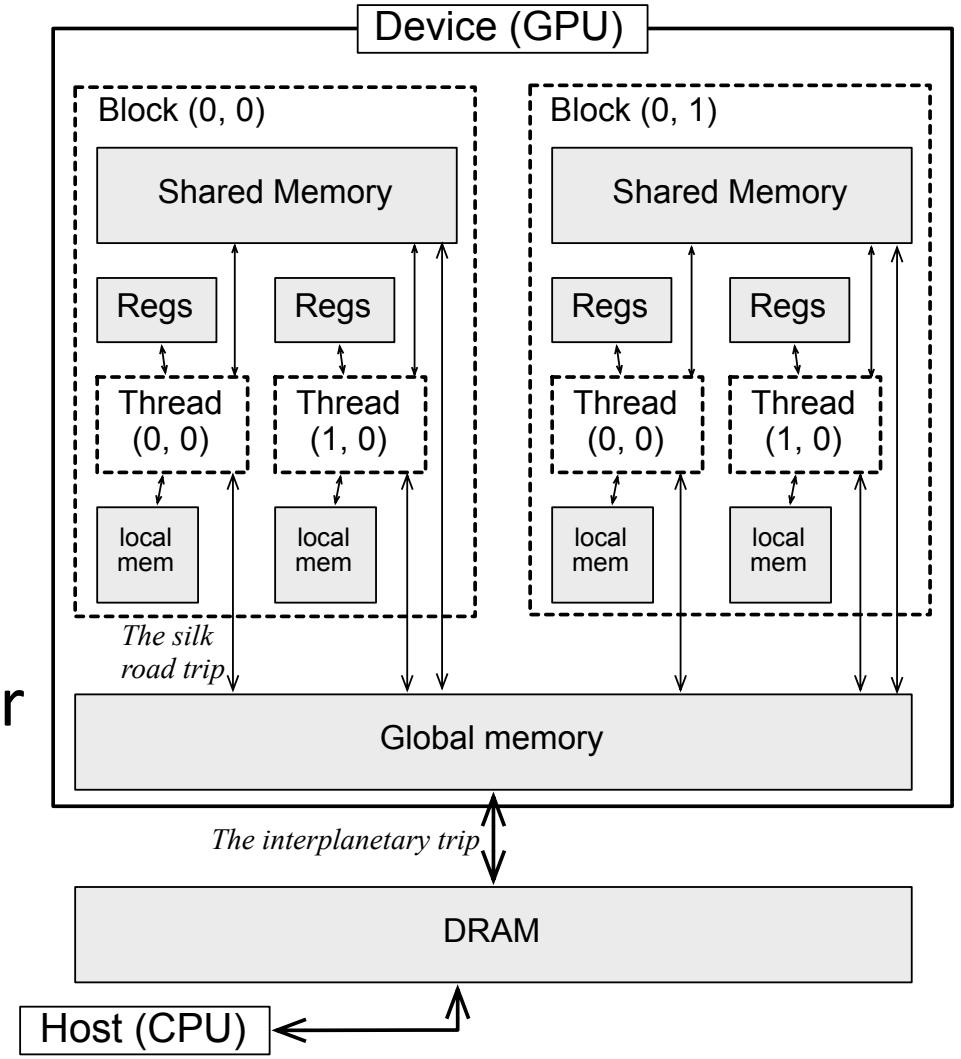
- **Registers**: fast, yet few. Private to each thread
- **Shared memory**: used by threads in the same block
- **Local memory**: off-chip and slow. Private to each thread
- **Global memory**: off-chip and slow. Used to provide communication between blocks and grids.

- Why can't we just leave all the data in registers?
- Why not leaving everything in the shared memory?

fernando@dcc.ufmg.br

The GPU Archive

- There are three main speed laws regarding memory access in GPUs. Can you guess them?
  - If you cannot, no worries: we will be talking about them later on... ☺



fernando@dcc.ufmg.br

The Traveler's Diary

The interplanetary trip	Commuting data between host and device
The silk road trip	Commuting data between global and shared memory
The bakery walk	Commuting data between shared memory and registers
Reaching a book on the table	Reading/Writing data to registers

The interplanetary trip

- **Copying data between GPU and CPU is pretty slow. CUDA provides some library functions for this:**
  - **`cudaMalloc`: allocates data in the GPU memory space**
  - **`cudaMemset`: fills a memory area with a value**
  - **`cudaFree`: frees the data in the GPU memory space**
  - **`cudaMemcpy`: copies data from CPU to GPU, or vice-versa**

fernando@dcc.ufmg.br

The interplanetary trip

```
int nbytes = 1024 * sizeof(int);
int *a_d = 0;
cudaMalloc( (void**) &a_d, nbytes);
cudaMemset( a_d, 0, nbytes);
cudaFree(a_d);
```

- What each of these calls to the CUDA API is doing?

fernando@dcc.ufmg.br

The interplanetary trip

```
int main(int argc, char** argv)  {
  float *a_h, *b_h;
  float *a_d, *b_d;
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  cudaMemset(&a_d, 0, nBytes);

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) { ASSERT( a_h[i] == b_h[i] );  }
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return EXIT_SUCCESS;
}
```

This program copies data from the host to the device, and then moves this data inside the device, and finally brings the data back to the host memory.

fernando@dcc.ufmg.br

The interplanetary trip

```
int main(int argc, char** argv)  {
  float *a_h, *b_h;
  float *a_d, *b_d;
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  cudaMemset(&a_d, 0, nBytes);

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) { ASSERT( a_h[i] == b_h[i] );  }
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return EXIT_SUCCESS;
}
```

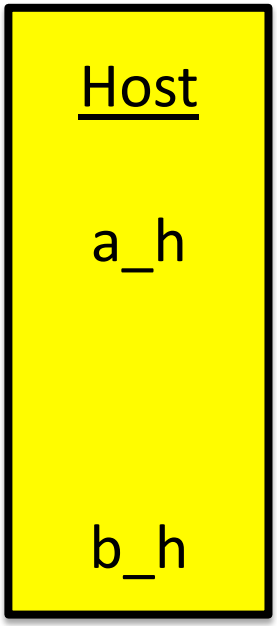
```
float *a_h, *b_h;
float *a_d, *b_d;
int N = 14, nBytes, i;
```

fernando@dcc.ufmg.br

The interplanetary trip

```
int main(int argc, char** argv) {
  float *a_h, *b_h;
  float *a_d, *b_d;
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  cudaMemset(&a_d, 0, nBytes);

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) { ASSERT( a_h[i] == b_h[i] ); }
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return EXIT_SUCCESS;
}
```

```
nBytes = N * sizeof(float);
a_h = (float*)malloc(nBytes);
b_h = (float*)malloc(nBytes);
```

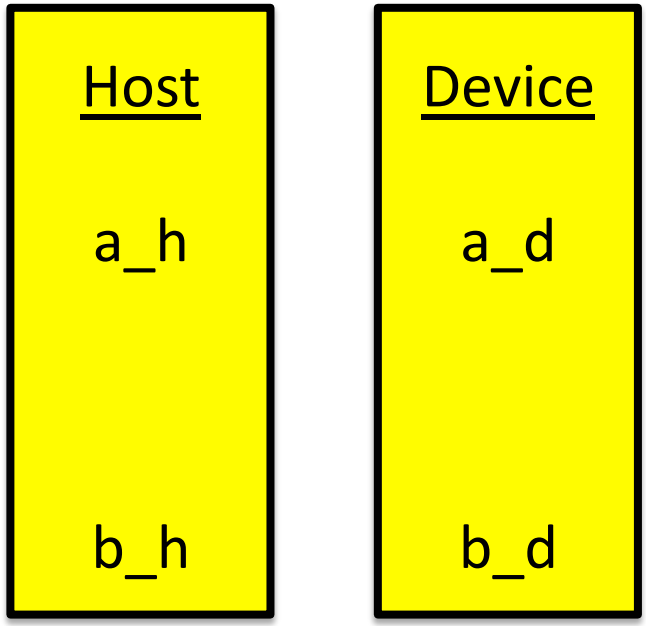


fernando@dcc.ufmg.br

The interplanetary trip

```
int main(int argc, char** argv) {
  float *a_h, *b_h;
  float *a_d, *b_d;
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  cudaMemset(&a_d, 0, nBytes);

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) { ASSERT( a_h[i] == b_h[i] ); }
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return EXIT_SUCCESS;
}
```

```
cudaMalloc((void**)&a_d, nBytes);
cudaMalloc((void**)&b_d, nBytes);
```

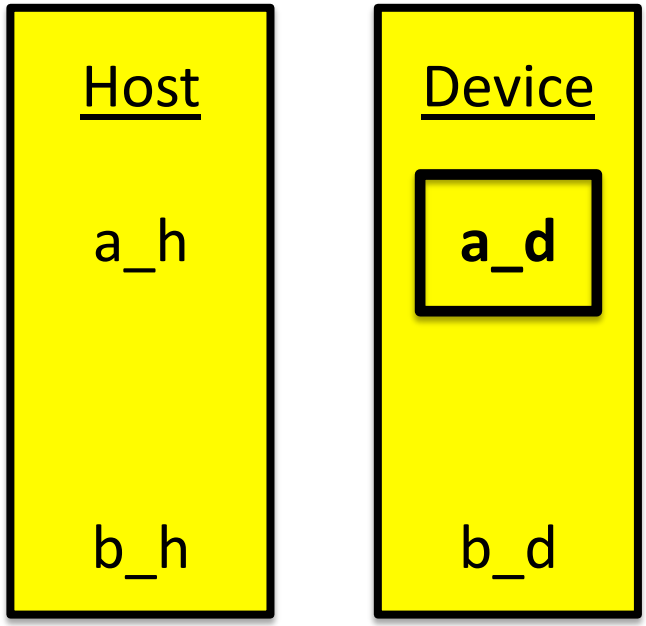


fernando@dcc.ufmg.br

The interplanetary trip

```
int main(int argc, char** argv) {
  float *a_h, *b_h;
  float *a_d, *b_d;
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  cudaMemset(&a_d, 0, nBytes);

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) { ASSERT( a_h[i] == b_h[i] ); }
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return EXIT_SUCCESS;
}
```

```
cudaMemset(&a_d, 0, nBytes);
```

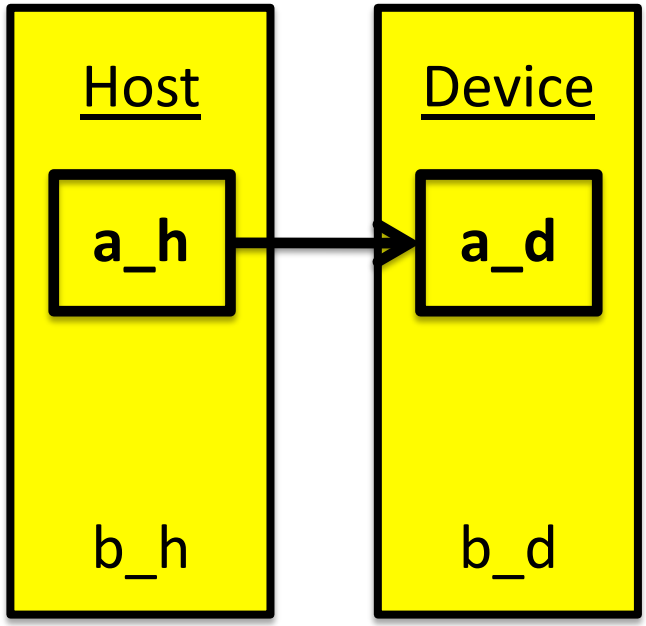


fernando@dcc.ufmg.br

The interplanetary trip

```
int main(int argc, char** argv) {
  float *a_h, *b_h;
  float *a_d, *b_d;
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  cudaMemset(&a_d, 0, nBytes);

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) { ASSERT( a_h[i] == b_h[i] ); }
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return EXIT_SUCCESS;
}
```

```
cudaMemcpy(a_d, a_h, nBytes,
  cudaMemcpyHostToDevice);
```

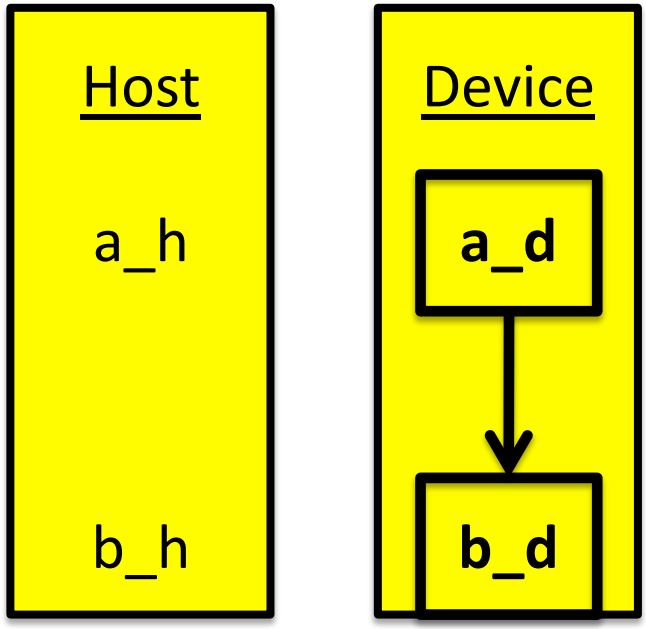


fernando@dcc.ufmg.br

The interplanetary trip

```
int main(int argc, char** argv) {
  float *a_h, *b_h;
  float *a_d, *b_d;
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  cudaMemset(&a_d, 0, nBytes);

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) { ASSERT( a_h[i] == b_h[i] ); }
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return EXIT_SUCCESS;
}
```

```
cudaMemcpy(b_d, a_d, nBytes,
  cudaMemcpyDeviceToDevice);
```

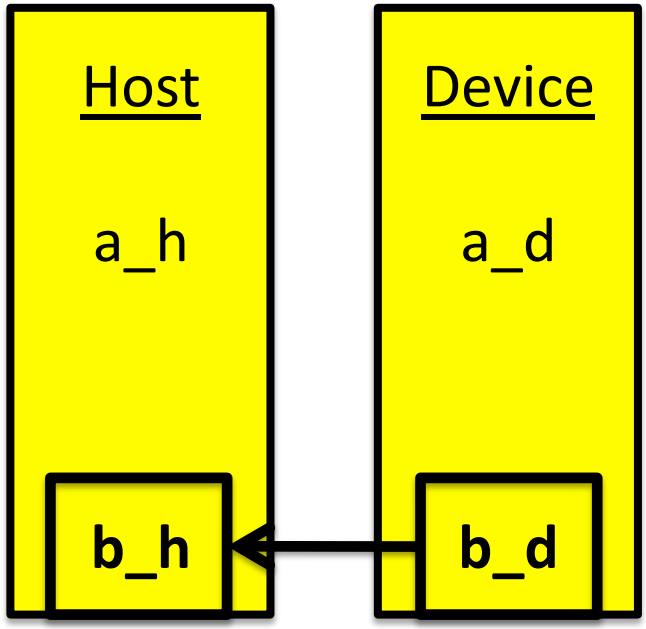


fernando@dcc.ufmg.br

The interplanetary trip

```
int main(int argc, char** argv) {
  float *a_h, *b_h;
  float *a_d, *b_d;
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  cudaMemset(&a_d, 0, nBytes);

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) { ASSERT( a_h[i] == b_h[i] ); }
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return EXIT_SUCCESS;
}
```

```
cudaMemcpy(b_h, b_d, nBytes,
  cudaMemcpyDeviceToHost);
```

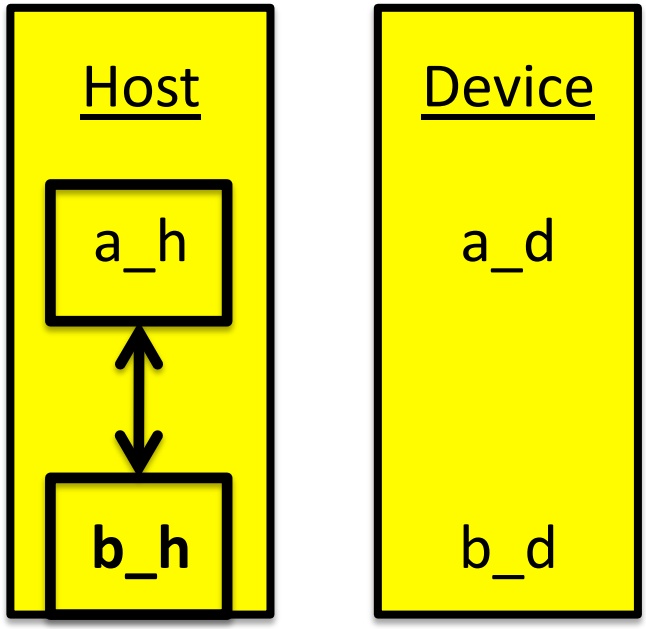


fernando@dcc.ufmg.br

The interplanetary trip

```
int main(int argc, char** argv) {
  float *a_h, *b_h;
  float *a_d, *b_d;
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  cudaMemset(&a_d, 0, nBytes);

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) { ASSERT( a_h[i] == b_h[i] ); }
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return EXIT_SUCCESS;
}
```

```
for (i=0; i< N; i++) {
  ASSERT(a_h[i] == b_h[i]);
}
```

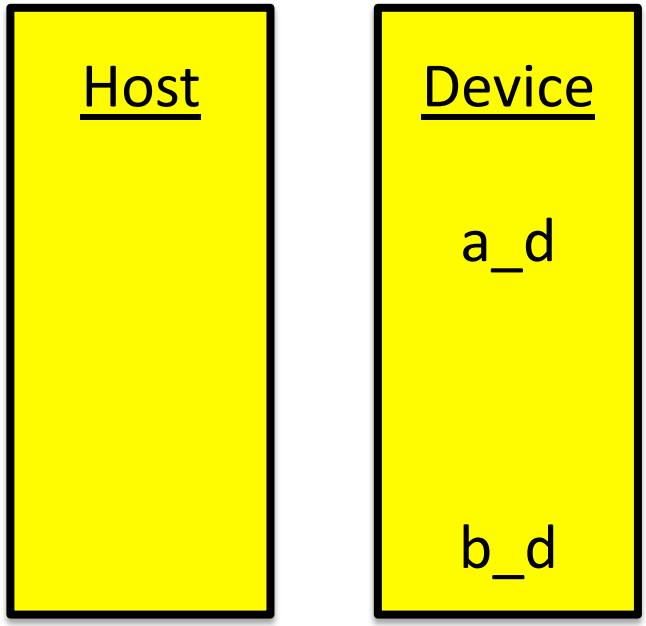


fernando@dcc.ufmg.br

The interplanetary trip

```
int main(int argc, char** argv) {
  float *a_h, *b_h;
  float *a_d, *b_d;
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  cudaMemset(&a_d, 0, nBytes);

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) { ASSERT( a_h[i] == b_h[i] ); }
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return EXIT_SUCCESS;
}
```

```
free(a_h); free(b_h);
```

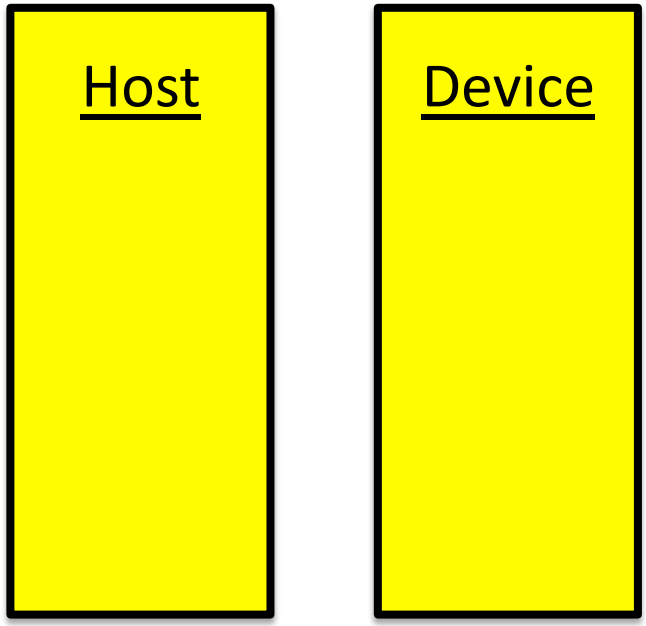


fernando@dcc.ufmg.br

The interplanetary trip

```
int main(int argc, char** argv)  {
  float *a_h, *b_h;
  float *a_d, *b_d;
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  cudaMemset(&a_d, 0, nBytes);

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) { ASSERT( a_h[i] == b_h[i] );  }
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return EXIT_SUCCESS;
}
```

```
cudaFree(a_d);
cudaFree(b_d);
```

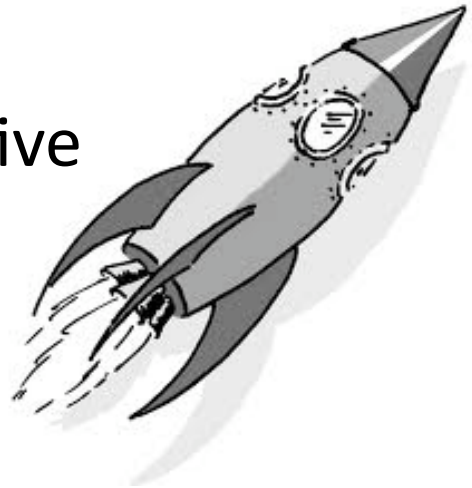


fernando@dcc.ufmg.br

First law of performance

- Inter-device communication, i.e, between the CPU and the GPU, should be minimized as much as possible.

- Inter-device communication is orders of magnitude slower than reading data from shared memory, for instance.
- That is why GPUs are not good for interactive applications.



fernando@dcc.ufmg.br

Avoid traveling whenever you can

```
cudaMalloc((void**) &d_vec, mem_size);

cudaMemcpy(d_vec, h_vec, mem_size,
cudaMemcpyHostToDevice);

kernel0<<< gridSize0, blockSize0 >>>(d_vec, vec_size);

kernel1<<< gridSize1, blockSize1 >>>(d_vec, vec_size);

cudaMemcpy(h_vec, d_vec, mem_size,
cudaMemcpyDeviceToHost);

cudaFree(d_vec);
```

`d_vec` **does not change** between kernel invocations.

No need to send it again.

From `maxSort`, available in the course webpage.

fernando@dcc.ufmg.br

Keep data on the GPU

- Once data is sent to the GPU, it stays on the DRAM, even after the kernel is done executing
- Try invoking kernels on data already on the GPU

- Can you think about a situation where you would have to leave a kernel, do some computation on the CPU, and then call another kernel?
- By the way, can you think about a problem that is inherently sequential?

fernando@dcc.ufmg.br

The GPU deserves complex work

- What is the complexity of copying data from the CPU to the GPU?
- Is it worth to do matrix sum in the GPU?
- Is it worth to do matrix multiplication in the GPU?

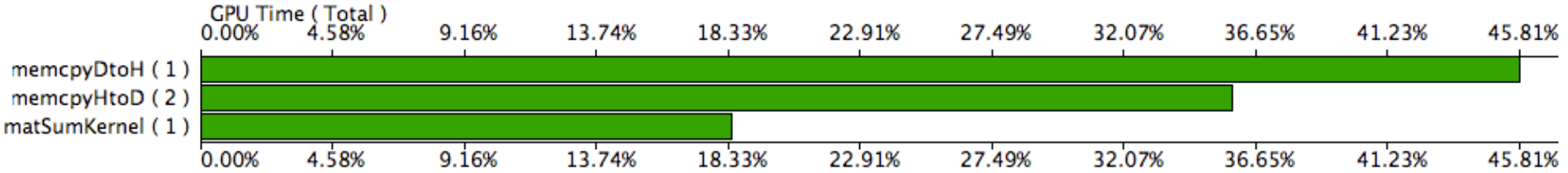
```
__global__
void matSumKernel(float* S, float*
A, float* B, int side) {
  int ij = tid.x + tid.y * side;
  A[ij] = B[ij] + C[ij];
}
```

```
__global__
void matMul1(float* B, float* C,
float* A, int w) {
  float v = 0.0;
  for (int k = 0; k < w; ++k) {
    v += B[tid.x*w+k] * C[k*w+tid.x];
  }
  A[tid.x + tid.y * w] = v;
}
```

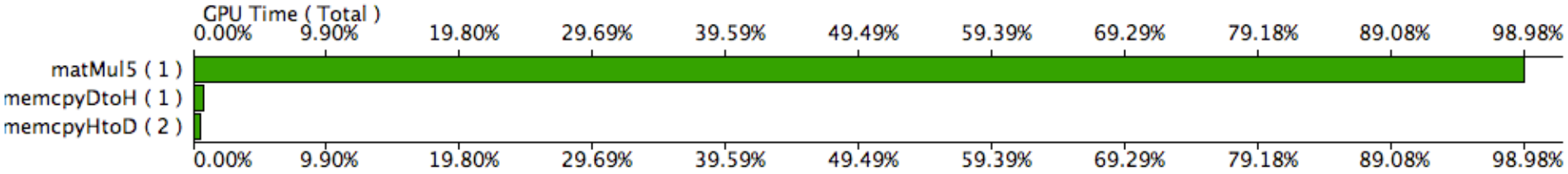
fernando@dcc.ufmg.br

Matrix Sum × Matrix Mul

- Matrix Sum:



- Matrix Mul:



fernando@dcc.ufmg.br

The ballerina's waltz

```
cudaMemcpy(dst, src, N * sizeof(float), dir);
kernel<<<N/nThreads, nThreads>>>(dst);
```

- Start working as soon as data is available - use a pipeline!

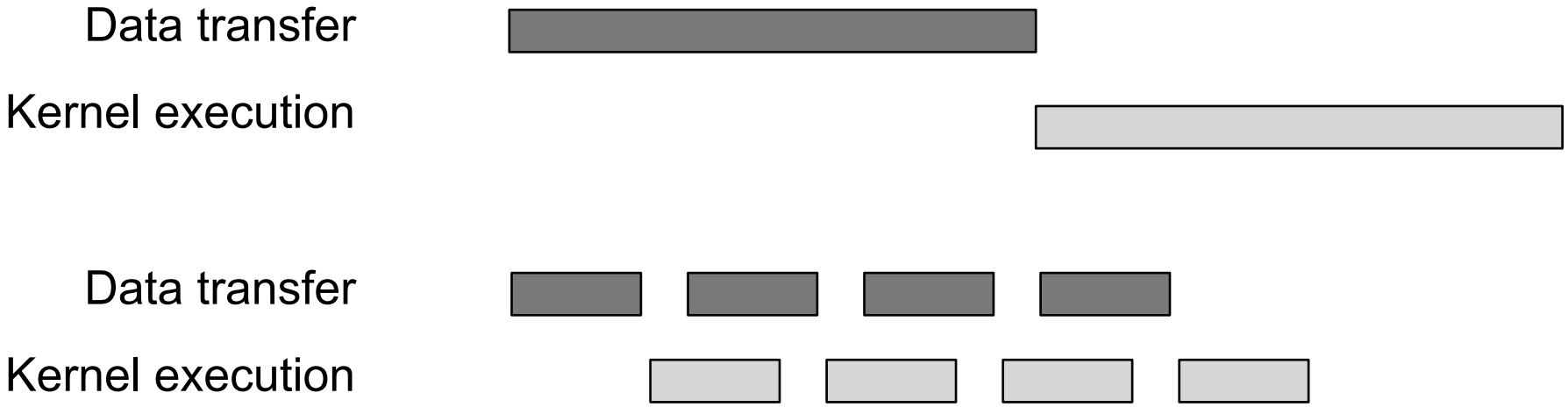
```
sz = N * sizeof(float) / nStreams;
for (i = 0; i < nStreams; i++) {
  offset = i * N / nStreams;
  cudaMemcpyAsync(dst+offset, src+offset, sz, dir, stream[i]);
}
for (i=0; i<nStreams; i++) {
  gridSize = N / (nThreads * nStreams);
  offset = i * N / nStreams;
  kernel<<<gridSize, nThreads, 0, stream[i]>>>(dst+offset);
}
```

- What is the glue between data and computation?

fernando@dcc.ufmg.br

The ballerina's waltz

- Asynchronous memory copy overlaps data transfer and GPU processing



- This technique to obtain parallelism is a pattern used in many different scenarios; could you name other examples?

The Silk Road Trip

- Reading or writing to the global memory is also slow.
  - But not as much as reading/writing between host and device.
  - The Global Memory is **on-board**.

fernando@dcc.ufmg.br

The Matrix (again)

- On a PRAM model what is the best complexity we can get to the matrix multiplication problem?
- What is the equivalent GPU kernel?

```
void matmult(float* B, float* C, float* A, int w) {
  for (unsigned int i = 0; i < w; ++i) {
    for (unsigned int j = 0; j < w; ++j) {
      A[i * w + j] = 0.0;
      for (unsigned int k = 0; k < w; ++k) {
        A[i * w + j] +=
          B[i * w + k] * C[k * w + j];
      }
    }
  }
}
```

fernando@dcc.ufmg.br

Matrix Multiplication Kernel

```
__global__ void matMul1(float* B, float* C, float* A, int Width)
{
  float Pvalue = 0.0;

  int tx = blockIdx.x * blockDim.x + threadIdx.x;
  int ty = blockIdx.y * blockDim.y + threadIdx.y;

  for (int k = 0; k < Width; ++k) {
    Pvalue += B[tx * Width + k] * C[k * Width + ty];
  }

  A[ty + tx * Width] = Pvalue;
}
```

- Given `Width` = 10, how many access to the global memory this program performs?
- How to know how many floating-point operations per second this program performs?

From `matMul`, available in the course webpage.

fernando@dcc.ufmg.br

GFLOPS

- How many instructions in the inner loop?
- How many floating point operations?
- GTX 8800 performs 172.8 G floating-point ops/sec
- What is the GFLOPS of our matrix multiplication kernel?
- **Yet measurements yield a lower number…**

```
 mov.f32        %f1, 0f00000000;
$Lt_0_1282:
  cvt.u64.u32   %rd3, %r7;
  mul.lo.u64    %rd4, %rd3, 4;
  ld.param.u64  %rd2, [B];
  add.u64       %rd5, %rd2, %rd4;
  ld.global.f32 %f2, [%rd5+0];
  cvt.u64.u32   %rd6, %r9;
  mul.lo.u64    %rd7, %rd6, 4;
  ld.param.u64  %rd1, [C];
  add.u64       %rd8, %rd1, %rd7;
  ld.global.f32 %f3, [%rd8+0];
  mad.f32       %f1, %f2, %f3, %f1;
  add.u32       %r7, %r7, 1;
  ld.param.s32  %r3, [w];
  add.u32       %r9, %r3, %r9;
  setp.ne.s32   %p2, %r7, %r8;
  @%p2 bra      $Lt_0_1282;
```

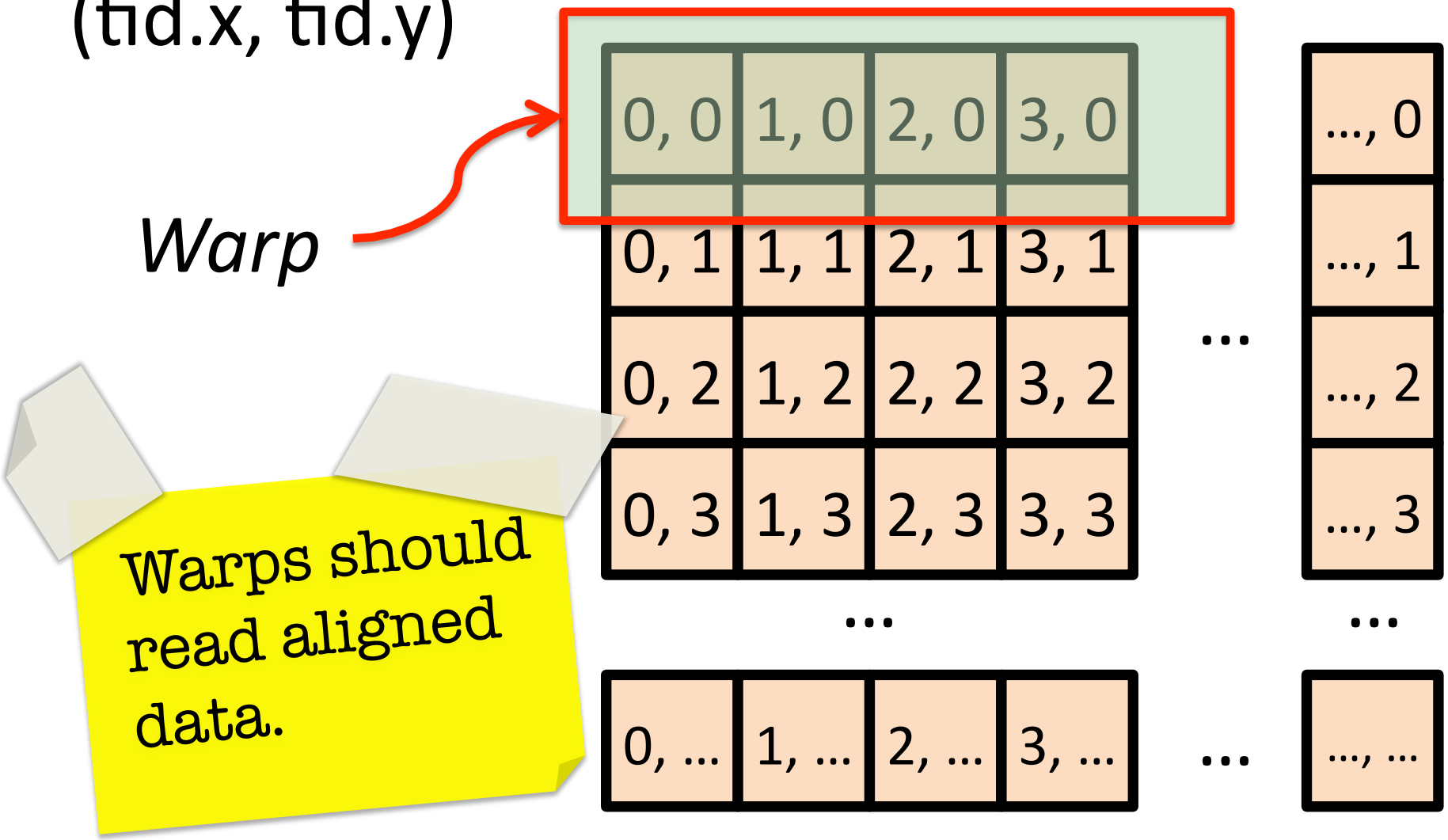
fernando@dcc.ufmg.br

The Tale of the Arab Merchant

- The global memory is divided into segments of 16 cells. If 16 threads read data from the same segment, the memory access contains only one trip.
- However, if each thread reads from a different segment…
- How to ensure that all the threads in a half-warp (16 threads) access the same segment of data?

fernando@dcc.ufmg.br

The Anatomy of a Block

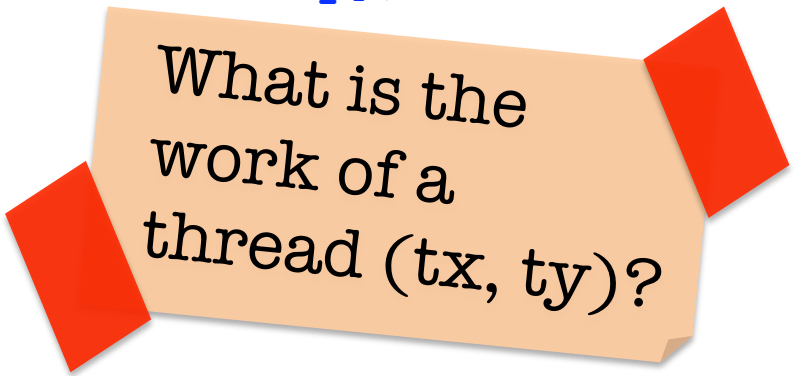


fernando@dcc.ufmg.br

Checking for coalesced access

- Is the program below doing coalesced memory access?

```
__global__ void matMul1(float* B, float* C, float* A, int Width)
{
  float Pvalue = 0.0;

  int tx = blockIdx.x * blockDim.x + threadIdx.x;
  int ty = blockIdx.y * blockDim.y + threadIdx.y;

  for (int k = 0; k < Width; ++k) {
    Pvalue += B[tx * Width + k] * C[k * Width + ty];
  }

  A[ty + tx * Width] = Pvalue;
}
```



fernando@dcc.ufmg.br

Checking Memory Accesses

```
...
int tx = blockIdx.x * blockDim.x + threadIdx.x;
int ty = blockIdx.y * blockDim.y + threadIdx.y;
for (int k = 0; k < Width; ++k) {
  Pvalue += B[tx * Width + k] * C[k * Width + ty];
}
A[ty + tx * Width] = Pvalue;
```

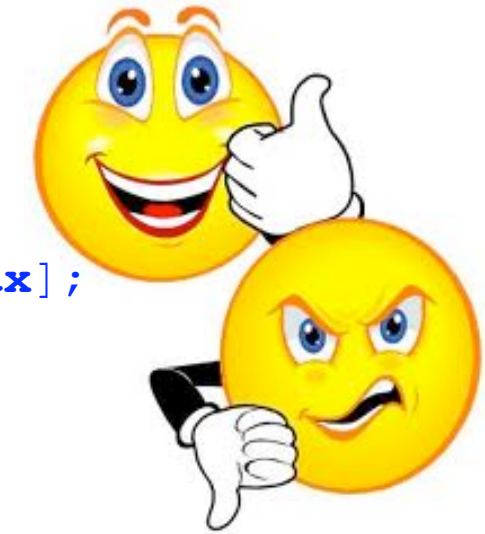
How can we improve this?

- Assuming `Width = 640` and `k = 0`
- Warp: (0, 0) (1, 0) (2, 0) … (31, 0)
- `B[tx * Width + k]`: B[0], B[640], B[1280], …, B[19840]
- `C[k * Width + ty]`: C[0], C[0] , C[0], …, C[0]
- `A[ty + tx * Width]`: A[0], A[640], A[1280], …, A[19840]

fernando@dcc.ufmg.br

Change the indexes a tiny bit…

```
__global__ void matMul5(float* B, float* C, float* A, int Width)
{
  float Pvalue = 0.0;

  int tx = blockIdx.x * blockDim.x + threadIdx.x;
  int ty = blockIdx.y * blockDim.y + threadIdx.y;

  for (int k = 0; k < Width; ++k) {
    Pvalue += B[ty * Width + k] * C[k * Width + tx];
  }

  A[tx + ty * Width] = Pvalue;
}
```



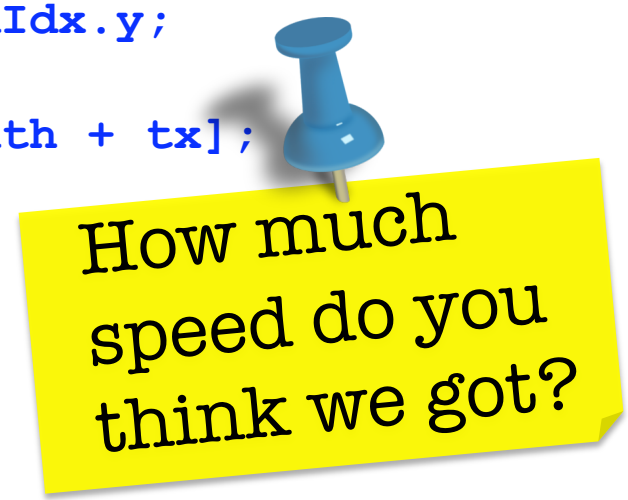
Contrast it with:

```
for (int k = 0; k < Width; ++k) {
  Pvalue += B[tx * Width + k] * C[k * Width + ty];
}
A[ty + tx * Width] = Pvalue;
```

fernando@dcc.ufmg.br

Why is it so much better?

```
...
int tx = blockIdx.x * blockDim.x + threadIdx.x;
int ty = blockIdx.y * blockDim.y + threadIdx.y;
for (int k = 0; k < Width; ++k) {
  Pvalue += B[ty * Width + k] * C[k * Width + tx];
}
A[tx + ty * Width] = Pvalue;
```



- Assuming `Width = 640` and `k = 0`
- Warp: (0, 0) (1, 0) (2, 0) … (31, 0)
- `B[ty * Width + k]`: B[0], B[0], B[0], …, B[0]
- `C[k * Width + tx]`: C[0], C[1] , C[2], …, C[31]
- `A[tx + ty * Width]`: A[0], A[1], A[2], …, A[31]

fernando@dcc.ufmg.br

Second law of performance

- Avoid going to the global or local memory. Ideally threads should be able to share as much data as possible in shared memory.

- Remember: the shared memory is *on-chip*; the global memory is *off-chip*
- Can we improve this matrix multiplication by sharing data among threads?

fernando@dcc.ufmg.br

How to apply the second law?

```
__global__ void matMul5(float* B, float* C, float* A, int Width)
{
  float Pvalue = 0.0;

  int tx = blockIdx.x * blockDim.x + threadIdx.x;
  int ty = blockIdx.y * blockDim.y + threadIdx.y;

  for (int k = 0; k < Width; ++k) {
    Pvalue += B[ty * Width + k] * C[k * Width + tx];
  }

  A[tx + ty * Width] = Pvalue;
}
```

- How many access to the global memory take place?
- Again: what is the shared memory?
- How to use the shared memory to improve this program?

```
__global__ void matMul6(float* B, float* C, float* A,
  int Width) {
  __shared__ float Bs[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Cs[TILE_WIDTH][TILE_WIDTH];
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int Row = blockIdx.x * TILE_WIDTH + tx;
  int Col = blockIdx.y * TILE_WIDTH + ty;
  float Pvalue = 0;
  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    Bs[ty][tx] = B[Col * Width + (m * TILE_WIDTH + tx)];
    Cs[ty][tx] = C[Row + (m * TILE_WIDTH + ty) * Width];
    __syncthreads();
    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += Bs[ty][k] * Cs[k][tx];
    __syncthreads();
  }
  A[Col * Width + Row] = Pvalue;
}
```

TILE_WIDTH
= blockDim.x
= blockDim.y

Why?

fernando@dcc.ufmg.br

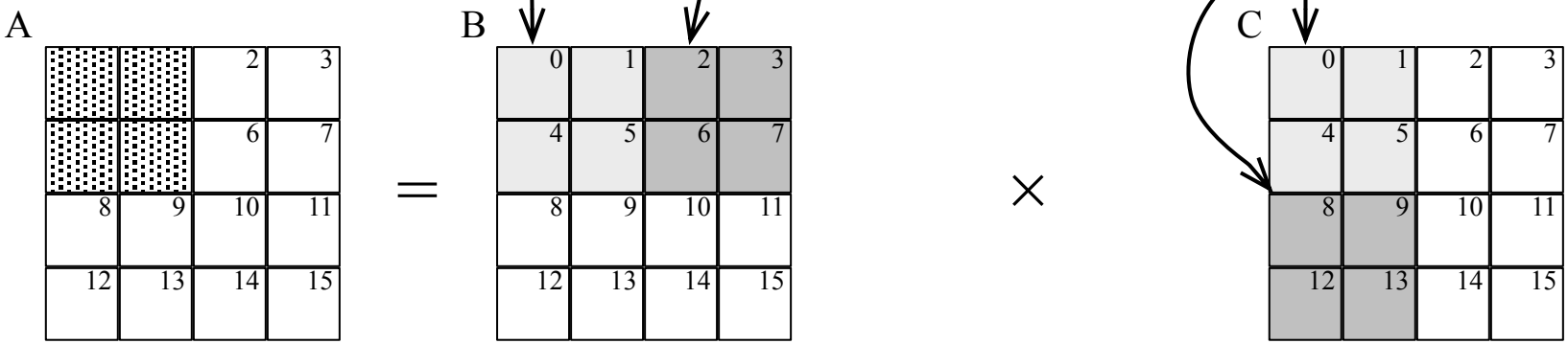
Tiling

Is the access coalesced or not?

Width = 4
TILE_WIDTH = 2

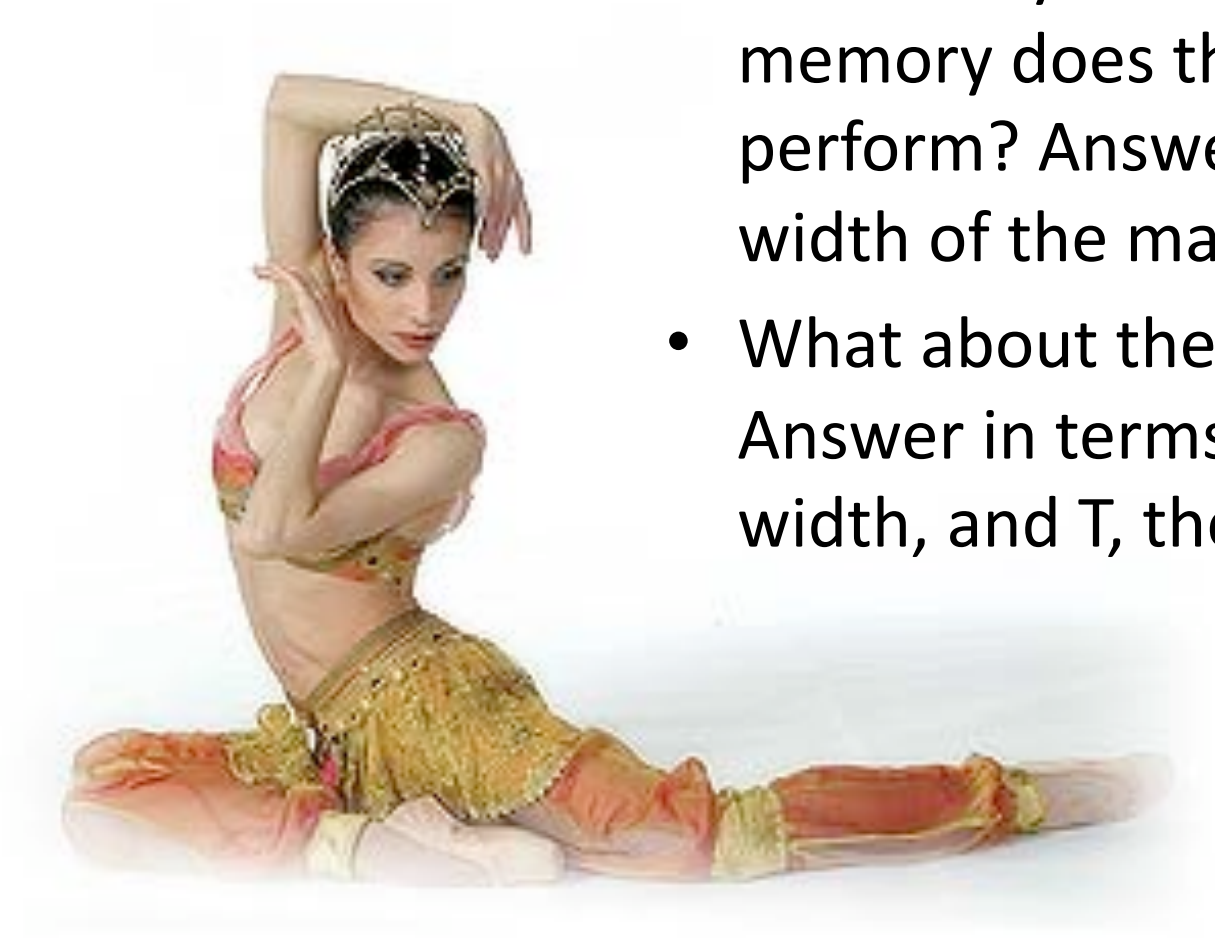
Col = blockIdx.x * TILE_WIDTH + tx
Row = blockIdx.y * TILE_WIDTH + ty

		Col * Width + (m * TILE_WIDTH + tx)		Row + (m * TILE_WIDTH + ty) * Width	
tx	ty	m = 0	m = 1	m = 0	m = 1
0	0	0	2	0	8
0	1	4	6	4	12
1	0	1	3	1	9
1	1	5	7	5	13



fernando@dcc.ufmg.br

The Tiles of Scheherazade

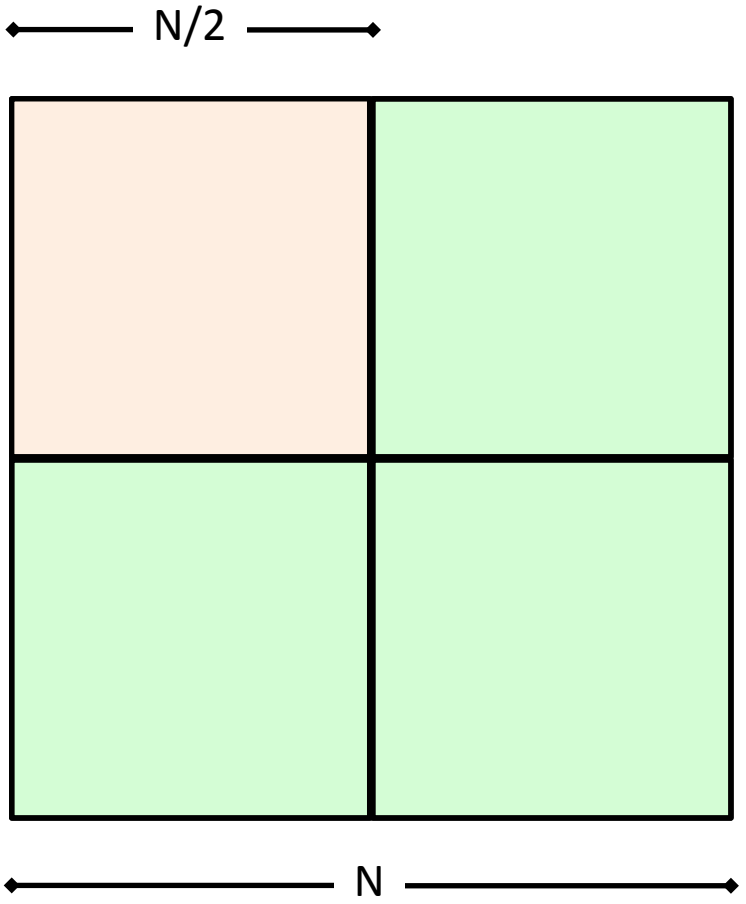


- How many access to the global memory does the original program perform? Answer in terms of N, the width of the matrices
- What about the tiled program? Answer in terms of N, the matrix width, and T, the tile width

fernando@dcc.ufmg.br

The Tiles of Scheherazade

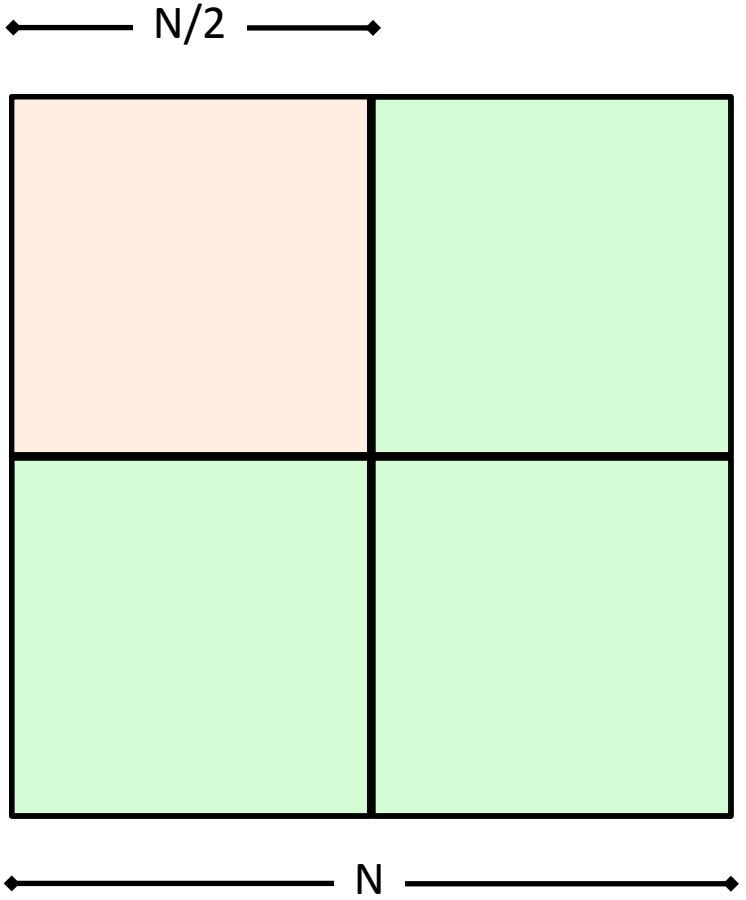
- How many reads with no tiling?
- How many reads with tiling?
  - Size of each tile:
  - Num of times each tile is read:
  - Num of tiles:



fernando@dcc.ufmg.br

The Tiles of Scheherazade

- How many reads with no tiling? **2N * N^2**
- How many reads with tiling?
  - Size of each tile:
  - Num of times each tile is read:
  - Num of tiles:

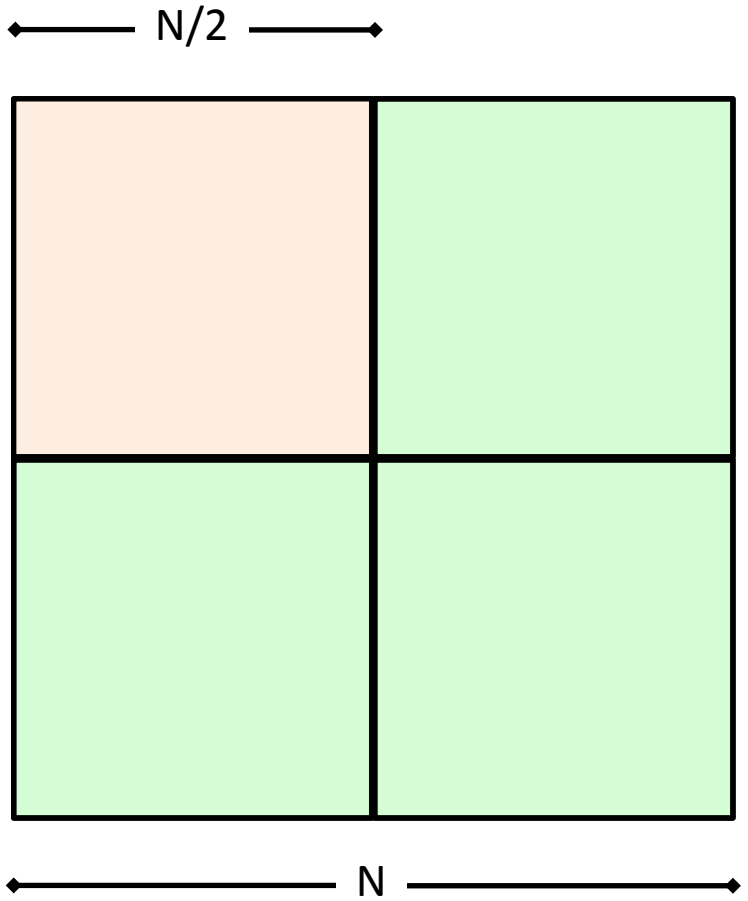


fernando@dcc.ufmg.br

The Tiles of Scheherazade

- How many reads with no tiling? **$2N * N^2$**
- How many reads with tiling?
  - Size of each tile: **$N^2/4$**
  - Num of times each tile is read: **$N/(N/2)$**
  - Num of tiles:

$(N/(N/2))^2 * 2$



So: $N^2/4 * 2 * 8 = 4*N^2$

fernando@dcc.ufmg.br

Back to the Future (Part II)

- Have you heard of pre-fetching?
- Can you apply pre-fetching on the tiled version of matrix multiplication?
  - Is it possible to overlap memory access and floating-point computation?
  - Can you think about some sort of pipelining?

- Where do we have the overlapping of data fetching and computation?
- Which operations inside the innermost loops are not floating-point operations?
- **How can we improve the code even more?**

```
float tmpC = C[Col + tx * Width];
float tmpB = B[Row * Width + ty];
int limit = Width/TILE_WIDTH;
int m = 0;
while(1) {
  Bs[tx][ty] = tmpB;
  Cs[tx][ty] = tmpC;
  __syncthreads();
  m++;
  if (m == limit) {
    #pragma unroll 1
    for (int k = 0; k < TILE_WIDTH; ++k) {
      Pvalue += Bs[tx][k] * Cs[k][ty];
    }
    __syncthreads();
    break;
  }
  tmpC = C[Col + (m*TILE_WIDTH + tx)*Width];
  tmpB = B[Row*Width + (m*TILE_WIDTH + ty)];
  for (int k = 0; k < TILE_WIDTH; ++k) {
    Pvalue += Bs[tx][k] * Cs[k][ty];
  }
  __syncthreads();
}
```

fernando@dcc.ufmg.br

Hollywood's Red Carpet

```
float Pvalue = 0;
for (int m = 0; m < w/8; ++m) {
  Mds[tx][ty] = Md[Row * w + (m * 8 + ty)];
  Nds[tx][ty] = Nd[Col + (m * 8 + tx) * w];
  __syncthreads();
  Pvalue += Mds[tx][ 0] * Nds[ 0][ty];
  Pvalue += Mds[tx][ 1] * Nds[ 1][ty];
  Pvalue += Mds[tx][ 2] * Nds[ 2][ty];
  Pvalue += Mds[tx][ 3] * Nds[ 3][ty];
  Pvalue += Mds[tx][ 4] * Nds[ 4][ty];
  Pvalue += Mds[tx][ 5] * Nds[ 5][ty];
  Pvalue += Mds[tx][ 6] * Nds[ 6][ty];
  Pvalue += Mds[tx][ 7] * Nds[ 7][ty];
  __syncthreads();
}
Pd[Row*Width+Col] = Pvalue;
```

fernando@dcc.ufmg.br

Lots of GFLOPS

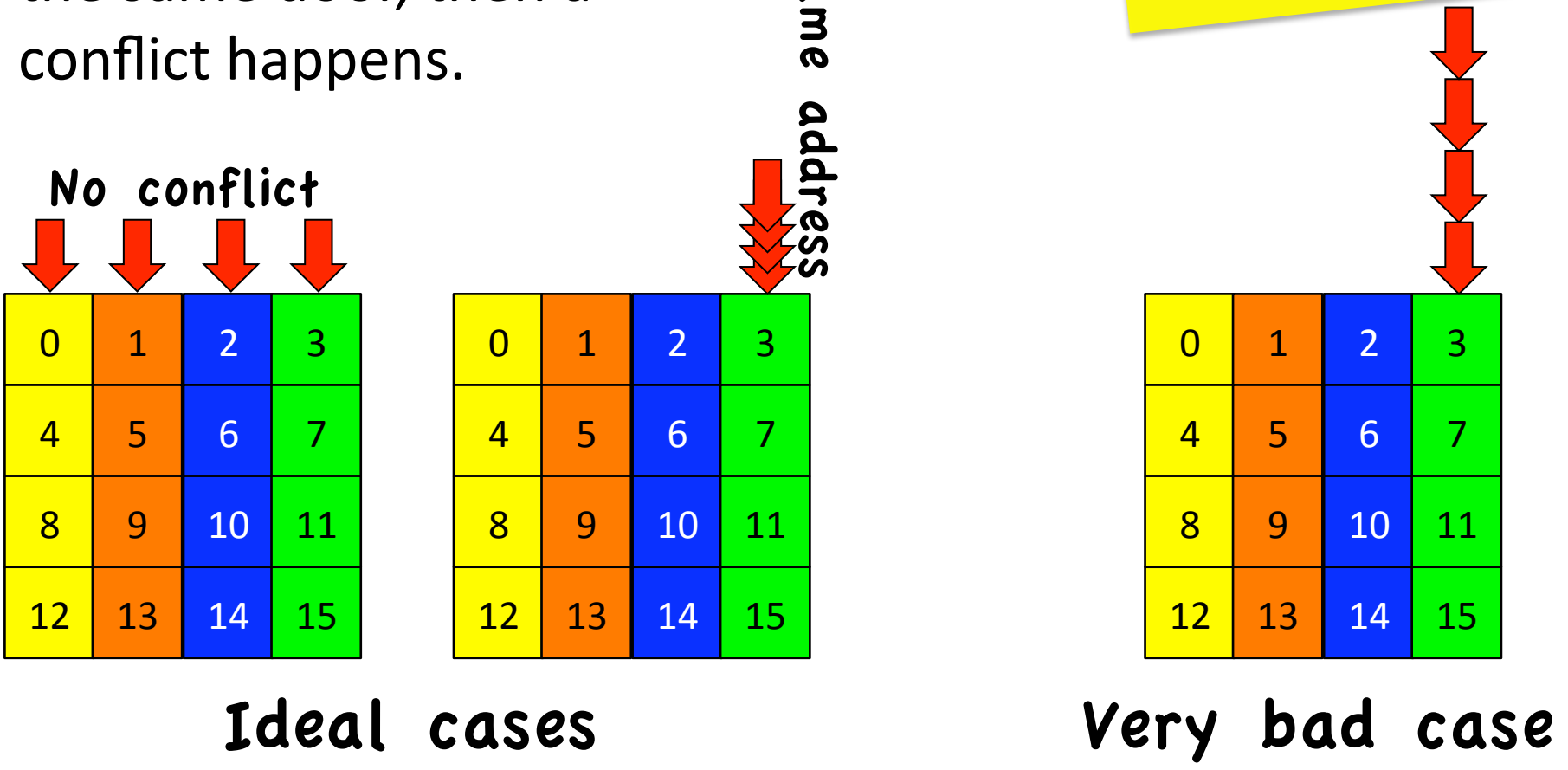
- What is the proportion of floating-point to non-floating-point operations in the innermost loop of the optimized program?
- What do we gain, at the hardware level, for having less branches to execute?
- About 1/2 of the instructions in the innermost loop are floating-point operations. What is the new GFLOP?

fernando@dcc.ufmg.br

Shared Bank Conflicts

- The shared memory has 16 access doors.
- If two threads of a half-warp read from the same door, then a conflict happens.

Lets assume 4 reading doors, and half-warp size = 8



From `transpose`, available in the course webpage. fernando@dcc.ufmg.br

Matrix transpose

BLOCK_WIDTH = 16

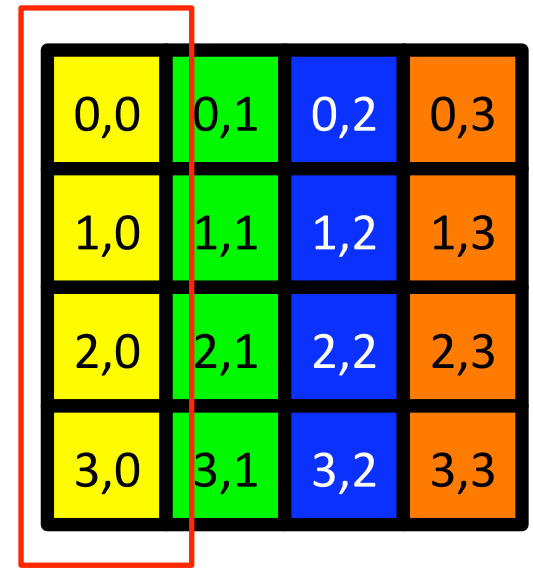
```
__global__ void transpose2(float* In, float* Out, int Width) {
  __shared__ float tile[BLOCK_WIDTH][BLOCK_WIDTH];
  // Compute the index of the data in the input matrix:
  int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
  int yIndex = blockIdx.y * blockDim.y + threadIdx.y;
  int index_in = yIndex * Width + xIndex;

  // Compute the index of the data in the output matrix:
  xIndex = blockIdx.y * blockDim.x + threadIdx.x;
  yIndex = blockIdx.x * blockDim.y + threadIdx.y;
  int index_out = yIndex * Width + xIndex;

  // Copy the data:
  tile[threadIdx.x][threadIdx.y] = In[index_in];
  __syncthreads();
  Out[index_out] = tile[threadIdx.y][threadIdx.x];
}
```

Is access to global memory coalesced?

Where is the bank conflict to access shared mem?

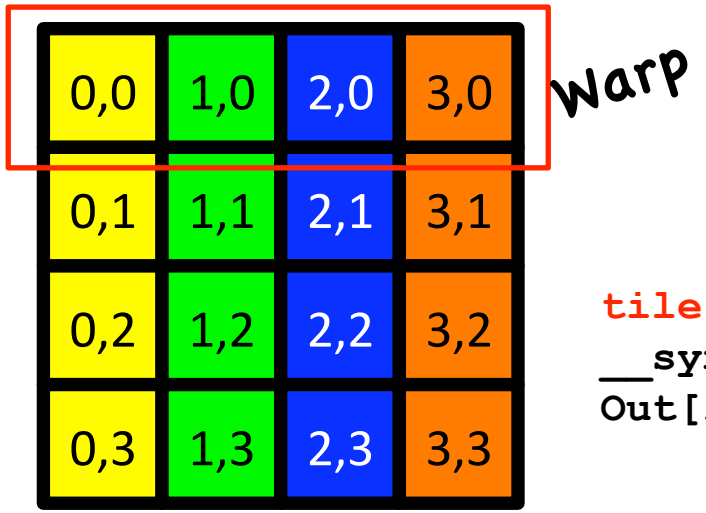
- **How can we solve the bank conflict?**

`tile[threadIdx.x][threadIdx.y]`



- Let's assume warp with 4 threads, and 4 access doors
- Each color is an access door
- The red square marks a warp
- If the warp falls on blocks having the same color, then we have a conflict

`tile[threadIdx.y][threadIdx.x]`



Ps.: in fig, (x,y) is warp index, not data index.

```
tile[threadIdx.x][threadIdx.y] = In[index_in];
__syncthreads();
Out[index_out] = tile[threadIdx.y][threadIdx.x];
```

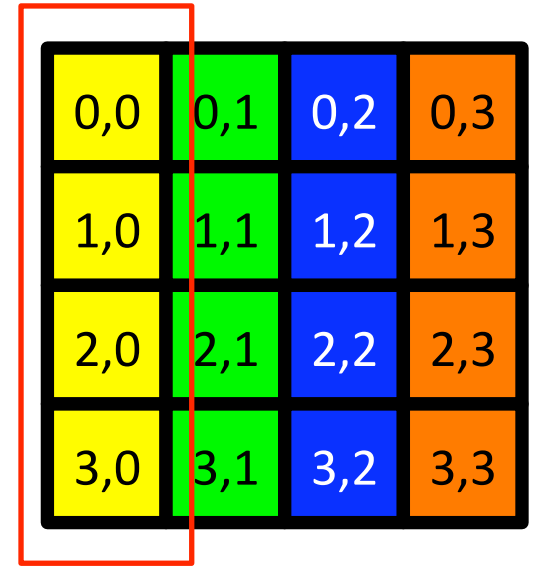
fernando@dcc.ufmg.br

A very simple solution:

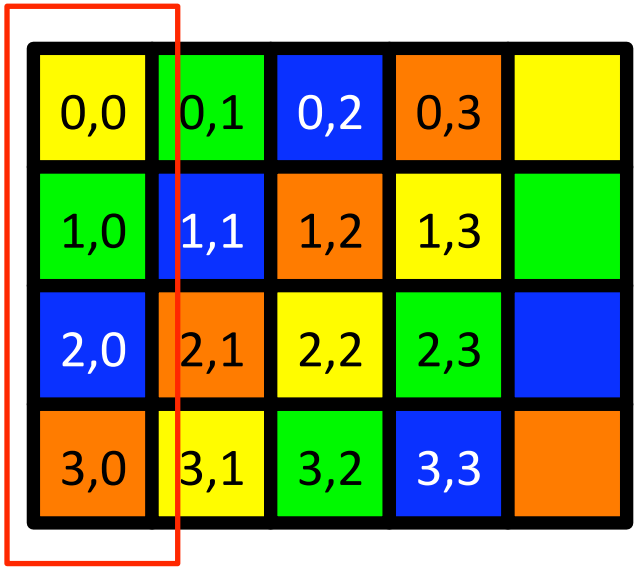
```
__global__ void transpose2(float* In, float* Out, int Width) {
  __shared__ float tile[BLOCK_WIDTH][BLOCK_WIDTH + 1];
  // Compute the index of the data in the input matrix:
  int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
  int yIndex = blockIdx.y * blockDim.y + threadIdx.y;
  int index_in = yIndex * Width + xIndex;

  // Compute the index of the data in the output matrix:
  xIndex = blockIdx.y * blockDim.x + threadIdx.x;
  yIndex = blockIdx.x * blockDim.y + threadIdx.y;
  int index_out = yIndex * Width + xIndex;

  // Copy the data:
  tile[threadIdx.x][threadIdx.y] = In[index_in];
  __syncthreads();
  Out[index_out] = tile[threadIdx.y][threadIdx.x];
}
```

- **You may not quite believe me…**

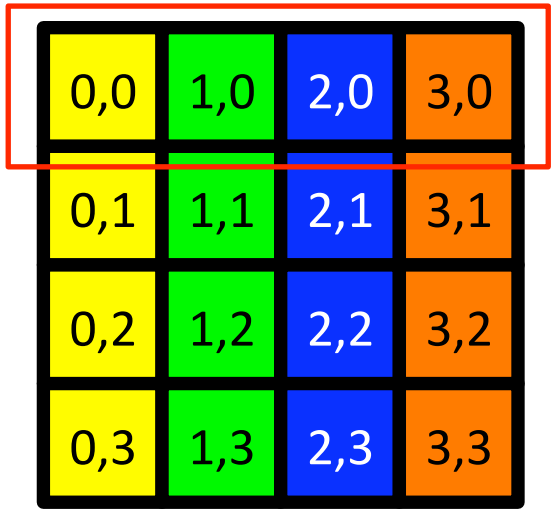
`tile[threadIdx.x][threadIdx.y]`

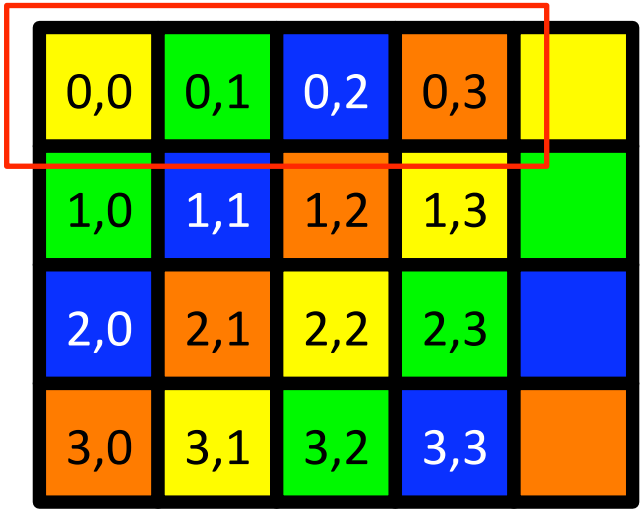


- The extra column breaks the alignment in both cases.



`tile[threadIdx.y][threadIdx.x]`





fernando@dcc.ufmg.br

Third Law of Performance

- The more registers threads use, the less threads fit in a block.

- The GTX 8800 has 8,192 registers per streaming processor (SP)
- A SP can run up to 768 threads
- Hence, we have 10 registers per thread

fernando@dcc.ufmg.br

Graphs to the rescue

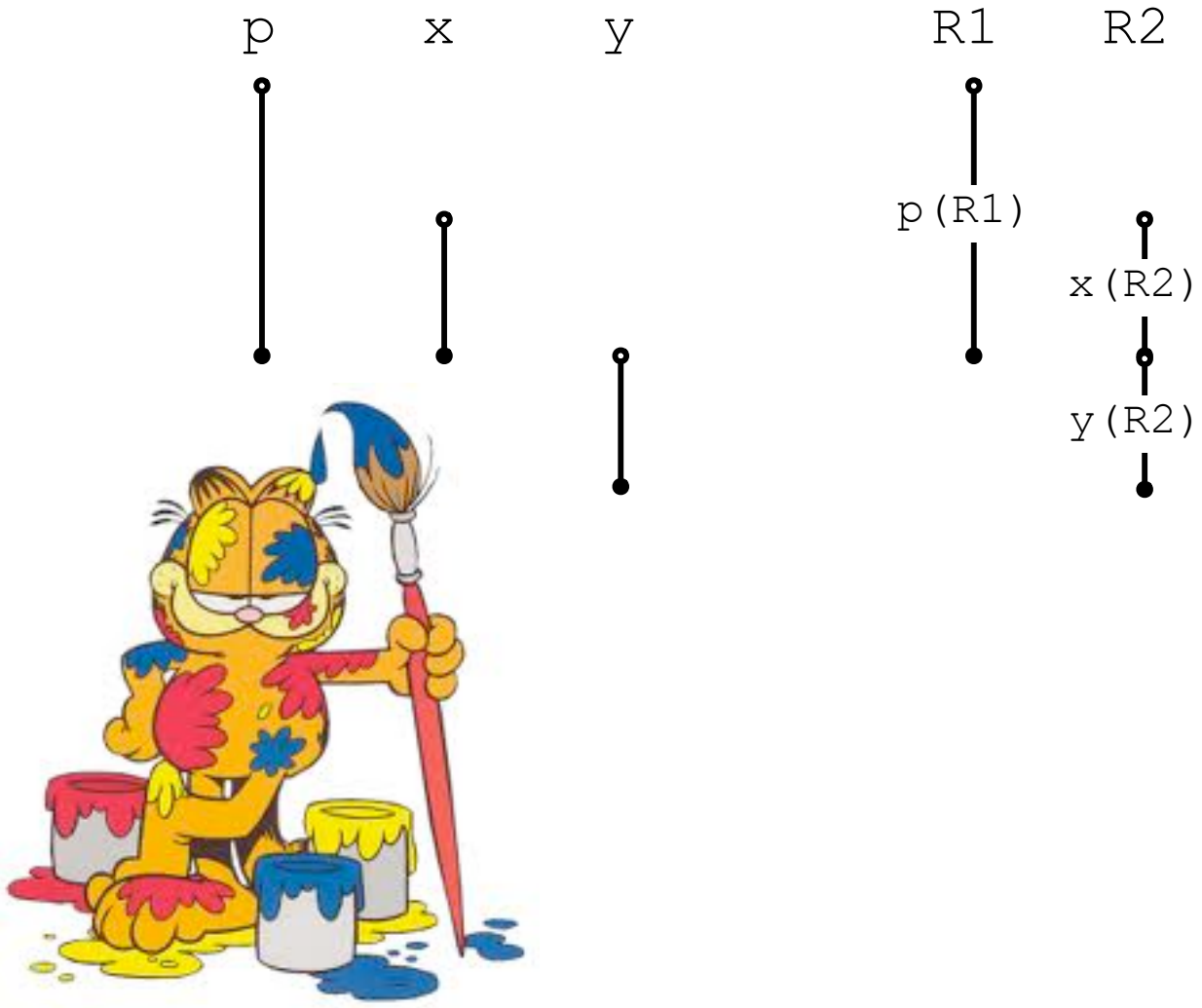
- How do we compute the register pressure in a given program?
  - For instance, what is the register pressure in the program below?

```
foo (int p) {
  x = p + 1
  y = x + p
  return y
}
```

fernando@dcc.ufmg.br

Register allocation via graph coloring

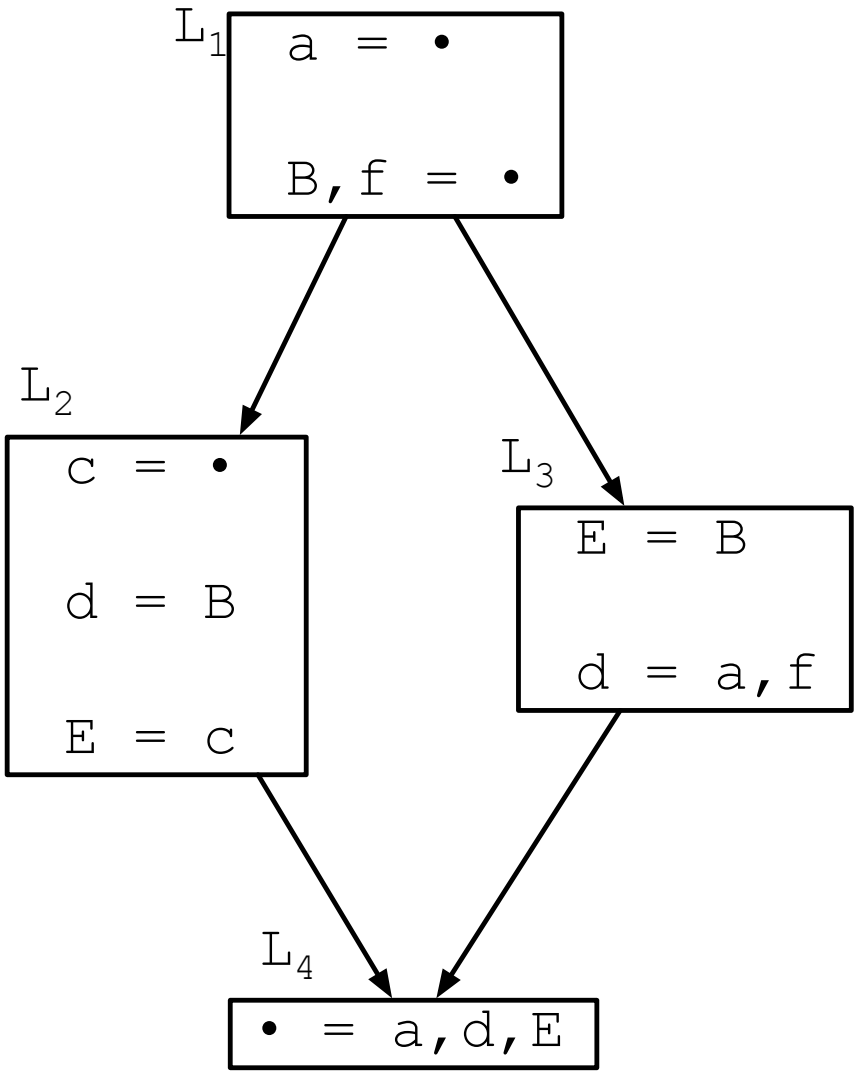
```
foo (int p) {
  x = p + 1
  y = x + p
  return y
}
```



fernando@dcc.ufmg.br

Register allocation is complicated

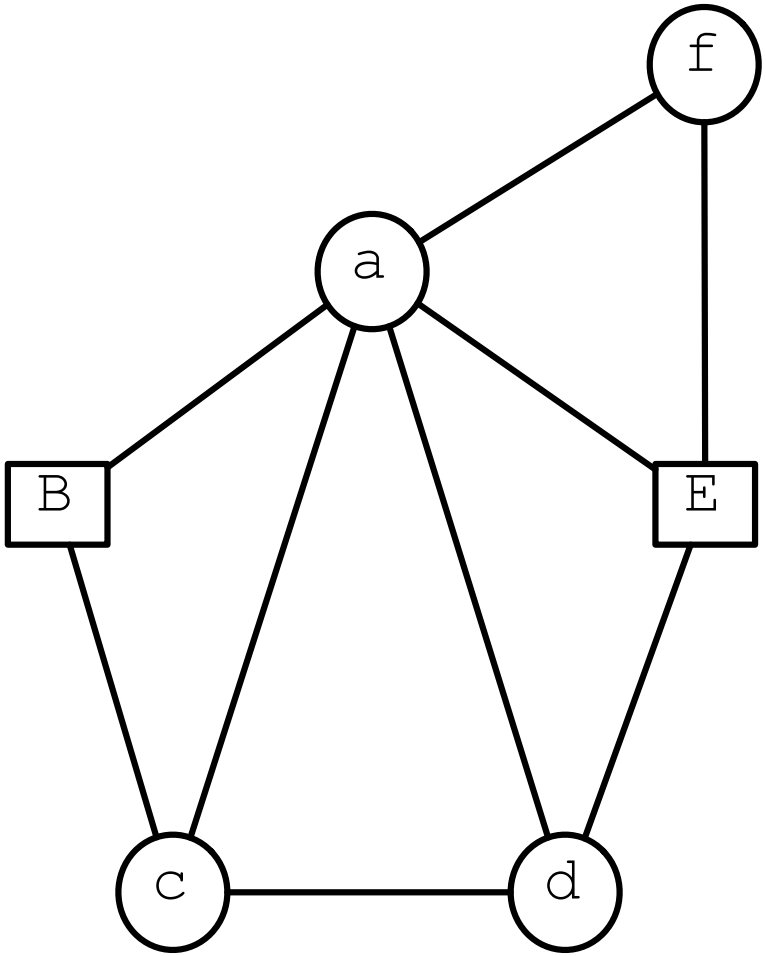
- Programs may have control flow.
  - How to produce the interference graph?
- Registers may have different sizes.



fernando@dcc.ufmg.br

Register allocation is complicated

- How hard is to find the register pressure?
- Can you think about algorithms other than graph coloring?



Example of high register pressure

```
__global__ void regPress1(float* In, float* Out, int Width) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if (tid < Width) {
    Out[tid] = 0.0;
    float a = 2.0F, b = 3.0F, c = 5.0F, d = 7.0F;
    for (int k = tid; k < Width * Width; k += Width) {
      Out[tid] += In[k] / (a - b);
      Out[tid] -= In[k] / (c - d);
      float aux = a;
      a = b;
      b = c;
      c = d;
      d = aux;
    }
  }
}
```

- How to check the register pressure of this program?
- How to lower its register pressure?

From `regPress`, available in the course webpage.

fernando@dcc.ufmg.br

Variable Sharing

```
__global__ void regPress2(float* In, float* Out, int Width) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if (tid < Width) {
    __shared__ float common0, common1;
    float c = 5.0F; float d = 7.0F;
    if (threadIdx.x == 0) { common0 = 2.0F; common1 = 3.0F; }
    __syncthreads();
    Out[tid] = 0.0;
    for (int k = tid; k < Width * Width; k += Width) {
      Out[tid] += In[k] / (common0 - common1);
      Out[tid] -= In[k] / (c - d);
      float aux = common0;
      if (threadIdx.x == 0) { common0 = common1; common1 = c; }
      __syncthreads();
      c = d; d = aux;
    }
  }
}
```

- Is there a systematic way to find all these "common" variables?

fernando@dcc.ufmg.br

In Short…

- Many classic optimizations apply on CUDA
  - Loop tiling
  - Loop unrolling
  - Pre-fetching
  - Register allocation
- But there are new optimizations too…
  - Stay tuned(!!!)