Linear Algebra on the GPU

Pawel Pomorski, *HPC Software Analyst*
SHARCNET, University of Waterloo

Overview

- Brief introduction to GPUs
- CUBLAS, CULA and MAGMA libraries
- Developing efficient linear algebra code for the GPU - case study of matrix transpose

#1 system on Fall 2012 TOP500 list - Titan

Oak Ridge National Labs - operational in October 2012
18,688 Opteron 16-core CPUs
18,688 NVIDIA Tesla K20 **GPUs**
17.6 peta FLOPS

Fell to #2 on Nov. 2013 list, beat by Intel Phi system

Why are GPUs fast?

Different paradigm, data parallelism (vs. task parallelism on CPU)

Stream processing, hardware capable of launching MANY threads

However, GPUs also have significant limitations and not all code can run fast on them

If a significant portion of your code cannot be accelerated, your speedup will be limited by Amdahl's Law

Low price of the CPU makes "supercomputing at home" possible

How to get running on the GPU?

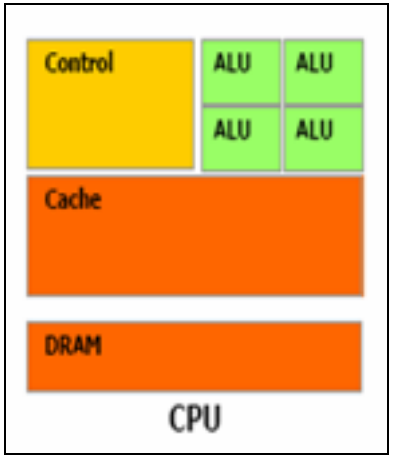
- Easiest case: the package you are using already has a GPU-accelerated version. No programming needed.
- Medium case: your program spends most of its time in library routines which have GPU accelerated versions. Use libraries that take advantage of GPU acceleration. Small programming effort required.
- Hard case: You cannot take advantage of the easier two possibilities, so you must convert some of your code to CUDA or OpenCL
- Newly available OpenACC framework is an alternative that should make coding easier.

Speedup

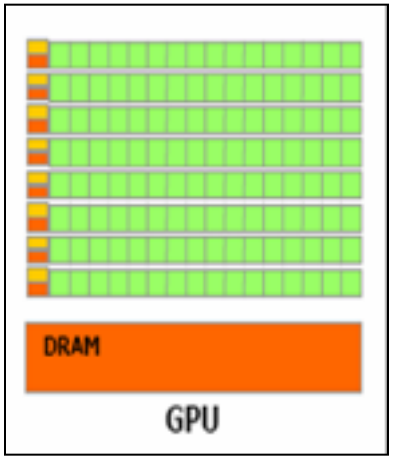
- What kind of speedup can I expect?
  - 0x – 2000x reported
  - 10x – 80x considered typical
  - >= 30x considered worthwhile (if coding required)
  - **5-10x speedup for fully optimized libraries**
- Speedup depends on
  - problem structure
    - need many identical independent calculations
    - preferably sequential memory access
  - level of intimacy with hardware
  - time investment

Comparing GPUs and CPUs

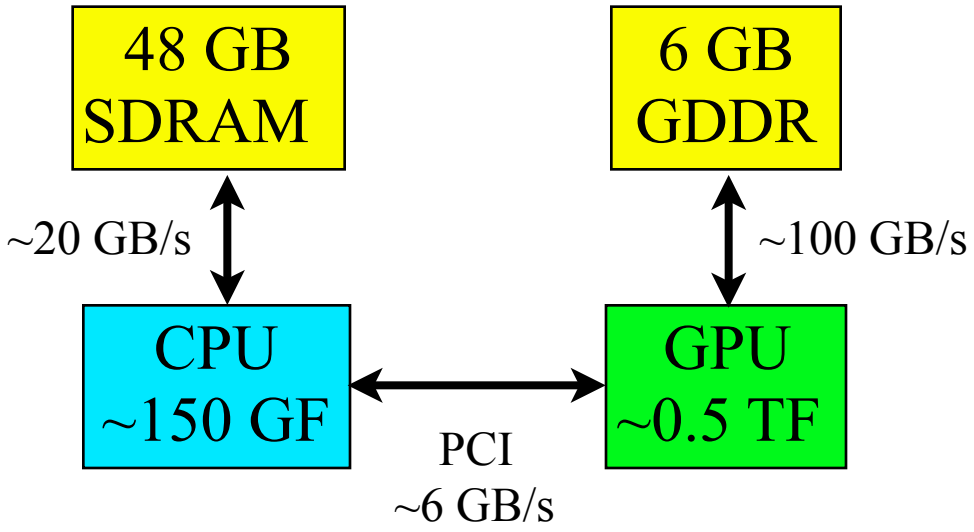
- CPU
  - "Jack of all trades"
  - task parallelism (diverse tasks)
  - minimize latency
  - multithreaded
  - some SIMD



- GPU
  - excel at number crunching
  - data parallelism (single task)
  - maximize throughput
  - super-threaded
  - large-scale SIMD



Be aware of memory bandwidth bottlenecks

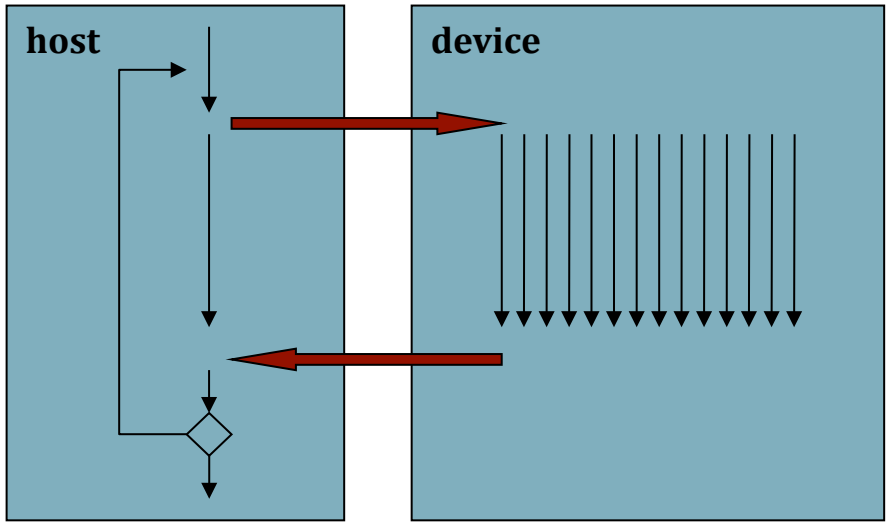


- The connection between CPU and GPU has low bandwidth
  - need to minimize data transfers
  - important to use asynchronous transfers if possible (overlap computation and transfer)
  - good idea to test bandwidth (with tool from SDK)

CUDA programming model

- The main CPU is referred to as the ***host***
- The compute device is viewed as a ***coprocessor*** capable of executing a large number of lightweight threads in parallel
- Computation on the device is performed by ***kernels***, functions executed in parallel on each data element
- Both the host and the device have their own ***memory***
  - the host and device cannot directly access each other's memory, but data can be transferred using the runtime API
- The host manages all memory allocations on the device, data transfers, and the invocation of kernels on the device

GPU as coprocessor



Kernel execution is asynchronous

Asynchronous memory transfers also available

- Basic paradigm
  - host uploads inputs to device
  - host remains busy while device performs computation
    - prepare next batch of data, process previous results, etc.
  - host downloads results
- Can be iterative or multi-stage

SHARCNET GPU systems

- Always check our software page for latest info! See also:
  https://www.sharcnet.ca/help/index.php/GPU_Accelerated_Computing

- *angel.sharcnet.ca*

  11 NVIDIA Tesla S1070 GPU servers —> **upgrade to GTX 750 Ti Maxwell cards in progress**

  each with 4 GPUs + 16GB of global memory

  each GPU server connected to ***two*** compute nodes (2 4-core Xeon CPUs + 8GB RAM each)

  1 GPU per quad-core CPU; 1:1 memory ratio between GPUs/CPUs

- visualization workstations

  Some old and don't support CUDA, but some have up to date cards, check list at:

  https://www.sharcnet.ca/my/systems/index

2012 arrival - “monk” cluster

- 54 nodes, InfiniBand interconnect, 80 Tb storage
- Node:
  8 x CPU cores (Intel Xeon 2.26 GHz)
  48 GB memory
  2 x M2070 GPU cards
- Nvidia Tesla M2070 GPU
  “Fermi” architecture
  ECC memory protection
  L1 and L2 caches
  2.0 Compute Capability
  448 CUDA cores
  515 Gigaflops (DP)

2014 arrival - “mosaic” cluster

- contributed system
- available end of 2014
- 20 nodes, each with:
  2 x Intel “Sandy Bridge” 10-core CPUs (20 cores total)
  256GB memory
  NVIDIA K20 GPU card

Language and compiler

- CUDA provides a set of extensions to the C programming language
  - new storage quantifiers, kernel invocation syntax, intrinsics, vector types, etc.
- CUDA source code saved in `.cu` files
  - host and device code and coexist in the same file
  - storage qualifiers determine type of code
- Compiled to object files using `nvcc` compiler
  - object files contain executable host and device code
- Can be linked with object files generated by other C/C++ compilers

Compiling

- nvcc -arch=sm_20 -O2 program.cu -o program.x
- -arch=sm_20 means code is targeted at Compute Capability 2.0 architecture (what monk has)
- -O2 optimizes the CPU portion of the program
- There are no flags to optimize CUDA code
- Various fine tuning switches possible
- SHARCNET has a CUDA environment module preloaded. See what it does by executing: module show cuda
- add -lcublas to link with CUBLAS libraries

Linear algebra on the GPU

- Linear algebra on the CPU: BLAS, LAPACK
- GPU analogues: CUBLAS, CULA, MAGMA
- CUSPARSE library for sparse matrices
- Use of highly optimized libraries is always better than writing your own code, especially since GPU codes cannot yet be efficiently optimized by compilers to achieve acceptable performance
- Writing efficient GPU code requires special care and understanding the peculiarities of underlying hardware

CUBLAS

- Implementation of BLAS (Basic Linear Algebra Subprograms) on top of CUDA
- Included with CUDA (hence free)
- Workflow:
  1. allocate vectors and matrices in GPU memory
  2. fill them with data
  3. call sequence of CUBLAS functions
  4. transfer results from GPU memory to host
- Helper functions to transfer data to/from GPU provided

Data layout

- for maximum compatibility with existing Fortran environments, CUBLAS library uses column-major storage and 1-based indexing (C/C++ uses row-major storage and 0-based indexing)
- use macro to convert

```
#define IDX2F(i,j,ld) ((((j)-1)*(ld)) + ((i)-1))
```

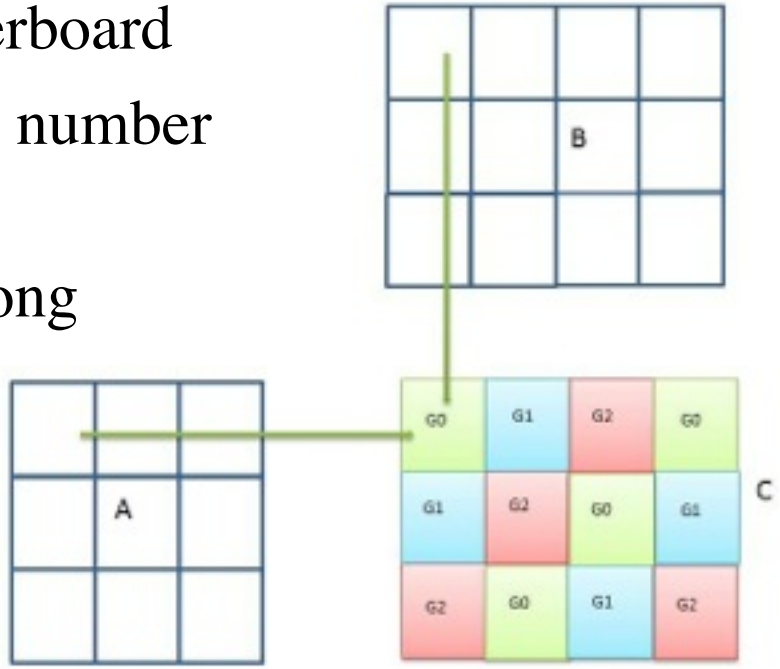
- CUBLAS library can be used by applications written in Fortran, via wrappers

CUBLAS in CUDA 4.0+

- new API, uses header file cublas_v2.h
- better suited to multi-thread and multi-GPU cases
- generally improved and simplified

CUBLAS in CUDA 6.0+

- multiple GPU support - cuBLAS-XT
- supports scaling across multiple GPUs connected to same motherboard
- linear performance scaling with number of GPUs
- matrix has to be distributed among the GPU memory spaces



Error checks

- in following example most error checks were removed for clarity
- each CUBLAS function returns a status object containing information about possible errors
- It's very important these objects to catch errors, via calls like this:

```
if (status != CUBLAS_STATUS_SUCCESS) {
print diagnostic information and exit}
```

Initialize program

```
/* Includes, system */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Includes, cuda */
#include <cuda_runtime.h>
#include <cublas_v2.h>

/* Matrix size */
#define N  (275)

/* Main */
int main(int argc, char** argv)
{
    cublasStatus_t status;
    float* h_A;
    float* h_B;
    float* h_C;
    float* d_A = 0;
    float* d_B = 0;
    float* d_C = 0;
    float alpha = 1.0f;
    float beta = 0.0f;
    int n2 = N * N;
    int i;
    cublasHandle_t handle;

    /* Initialize CUBLAS */

    status = cublasCreate(&handle);
```

Allocate and initialize memory on CPU/GPU

```
/* Allocate host memory for the matrices */
   h_A = (float*)malloc(n2 * sizeof(h_A[0]));
   h_B = (float*)malloc(n2 * sizeof(h_B[0]));

   /* Fill the matrices with test data */
   for (i = 0; i < n2; i++) {
       h_A[i] = rand() / (float)RAND_MAX;
       h_B[i] = rand() / (float)RAND_MAX;
  }

   /* Allocate device memory for the matrices */
   if (cudaMalloc((void**)&d_A, n2 * sizeof(d_A[0])) != cudaSuccess) {
       fprintf (stderr, "!!!! device memory allocation error (allocate A)\n");
       return EXIT_FAILURE;
   }
   if (cudaMalloc((void**)&d_B, n2 * sizeof(d_B[0])) != cudaSuccess) {
       fprintf (stderr, "!!!! device memory allocation error (allocate B)\n");
       return EXIT_FAILURE;
   }
   if (cudaMalloc((void**)&d_C, n2 * sizeof(d_C[0])) != cudaSuccess) {
       fprintf (stderr, "!!!! device memory allocation error (allocate C)\n");
       return EXIT_FAILURE;
   }

  /* Initialize the device matrices with the host matrices */
   status = cublasSetVector(n2, sizeof(h_A[0]), h_A, 1, d_A, 1);
   status = cublasSetVector(n2, sizeof(h_B[0]), h_B, 1, d_B, 1);
```

Call main CUBLAS function, get result

```
    /* Performs operation using cublas */
    status = cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, N, N, &alpha, d_A, N, d_B, N,
&beta, d_C, N);

    /* Allocate host memory for reading back the result from device memory */
    h_C = (float*)malloc(n2 * sizeof(h_C[0]));

    /* Read the result back */
    status = cublasGetVector(n2, sizeof(h_C[0]), d_C, 1, h_C, 1);
```

Cleanup

```
   /* Memory clean up */
    free(h_A);
    free(h_B);
    free(h_C);
    if (cudaFree(d_A) != cudaSuccess) {
        fprintf (stderr, "!!!! memory free error (A)\n");
        return EXIT_FAILURE;
    }
    if (cudaFree(d_B) != cudaSuccess) {
        fprintf (stderr, "!!!! memory free error (B)\n");
        return EXIT_FAILURE;
    }
    if (cudaFree(d_C) != cudaSuccess) {
        fprintf (stderr, "!!!! memory free error (C)\n");
        return EXIT_FAILURE;
    }

    /* Shutdown */
    status = cublasDestroy(handle);


    return EXIT_SUCCESS;
}
```

CUBLAS performance - matrix multiplication

- multiplying single-precision (SP)/double precision (DP) matrices (16384 x 32) and (32 x 16384), result is a (16384 x 16384) matrix
- CPU Intel MKL library (11.1.4) - single core and threaded
- CUBLAS library (CUDA 5.0 version)
- GPU timing results with and without memory transfer
- tested on monk M2070 Tesla
- NOTE: non-Tesla cards generally offer poor DP performance (DP calculations 10x slower)

CUBLAS performance - matrix multiplication

	SP time (s)	DP time (s)
CPU - 1 core	1.31	2.71
CPU - 8 core	0.42	0.92
CUBLAS - without memory transfer	0.041	0.069
CUBLAS - with memory transfer	0.22	0.42

CUBLAS performance - matrix multiplication

	SP time (s)	DP time (s)
CPU - 1 core	0.32x	0.33x
CPU - 8 core	1.0x	1.0x
CUBLAS - without memory transfer	10.2x	13.3x
CUBLAS - with memory transfer	1.9x	2.2x

CUBLAS batching kernels

- requires Fermi architecture, which allows to execute multiple kernels simultaneously
- can use streams to batch execution of small kernels
- useful for cases where computing many smaller matrices needed

CUSPARSE

- included with CUDA (free)
- set of basic linear algebra subroutines for handling sparse matrices on top of CUDA
- analogue of CUBLAS, but more basic in functionality

CULA

- Implementation of LAPACK (Linear Algebra PACKage) for CUDA-enabled NVIDIA GPUs
- only basic version (with limited number of functions implemented) bundled for free with CUDA
- premium version not free, individual licenses no longer available
- No license currently available on SHARCNET
- Has its own (improved) BLAS
- Examples in /opt/sharcnet/cuda/version/cula

Interfaces

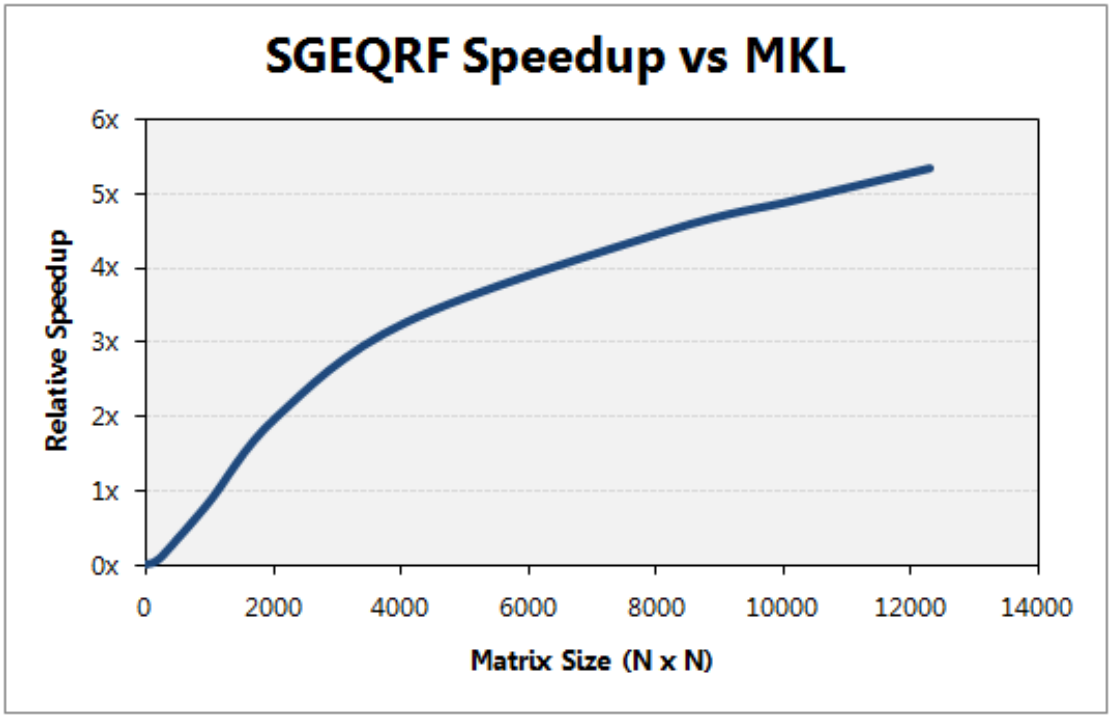
- **Standard** - functions work on data in main memory, GPU completely hidden
- **Device** - follows CUBLAS interface, user allocates and populates GPU memory
- **Fortran** - like Standard but in Fortran
- **Link** - for porting existing linear algebra codes to GPU, matches function names to several popular algebra packages. Provides fallback to CPU execution if user does not have GPU or problem too small to use GPU efficiently

CULA features

- uses column major ordering
- functions take arguments by value instead of reference
- functions use fewer variables

```
// Common variables
...

// CULA
culaStatus s;
s = culaSgesvd('O', 'N', m, m, a, lda, s, u, ldu, vt, ldvt);

// Traditional
char jobu = 'O';
char jobvt = 'N';
int info;
sgesvd(&jobu, &jobvt, &m, &n, a, &lda, s, u, &ldu, vt, &ldvt, &info);
```

Expected performance



Error catching function

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <culapack.h>


void checkStatus(culaStatus status)
{
    char buf[80];

    if(!status)
        return;

    culaGetErrorInfoString(status, culaGetErrorInfo(), buf, sizeof(buf));
    printf("%s\n", buf);

    culaShutdown();
    exit(EXIT_FAILURE);
}
```

Initialize

```
int main(int argc, char** argv)
{
    int M = 8192;
    int N = M;

    culaStatus status;

    float* A = NULL;
    float* TAU = NULL;

    printf("Allocating Matrices\n");
    A = (float*)malloc(M*N*sizeof(float));
    TAU = (float*)malloc(N*sizeof(float));
    if(!A || !TAU)
        exit(EXIT_FAILURE);

    printf("Initializing CULA\n");
    status = culaInitialize();
    checkStatus(status);
```

Call LAPACK function

```
    memset(A, 0, M*N*sizeof(float));

    printf("Calling culaSgeqrf\n");
    status = culaSgeqrf(M, N, A, M, TAU);
    checkStatus(status);

    printf("Shutting down CULA\n");
    culaShutdown();

    free(A);
    free(TAU);

    return EXIT_SUCCESS;
}
```

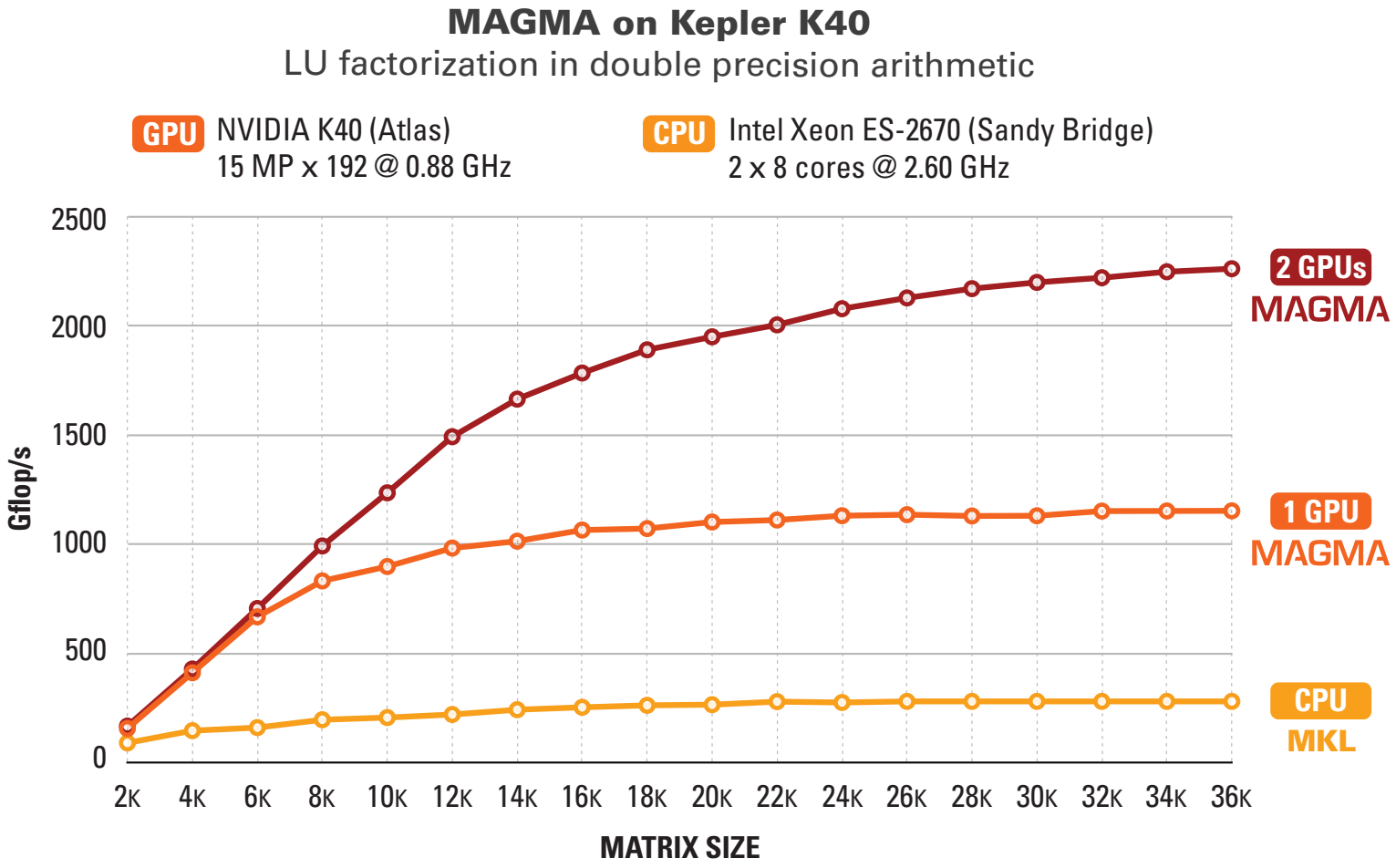
CULA on multiple GPUs

- currently default mode is running on one GPU
- considered thread-safe so can be used on multiple GPUs when launched in different threads running on different GPUs
- pCULA library for multiple GPUs available since 2012

MAGMA library

- The MAGMA project aims to develop a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures, starting with current "Multicore+GPU" systems.
- http://icl.cs.utk.edu/magma/
- free and open source
- implementations for CPU, NVIDIA GPU, Intel Phi, OpenCL
- check MAGMA documentation for which routines implemented where
- multi-GPU support for certain routines

MAGMA library



MAGMA example

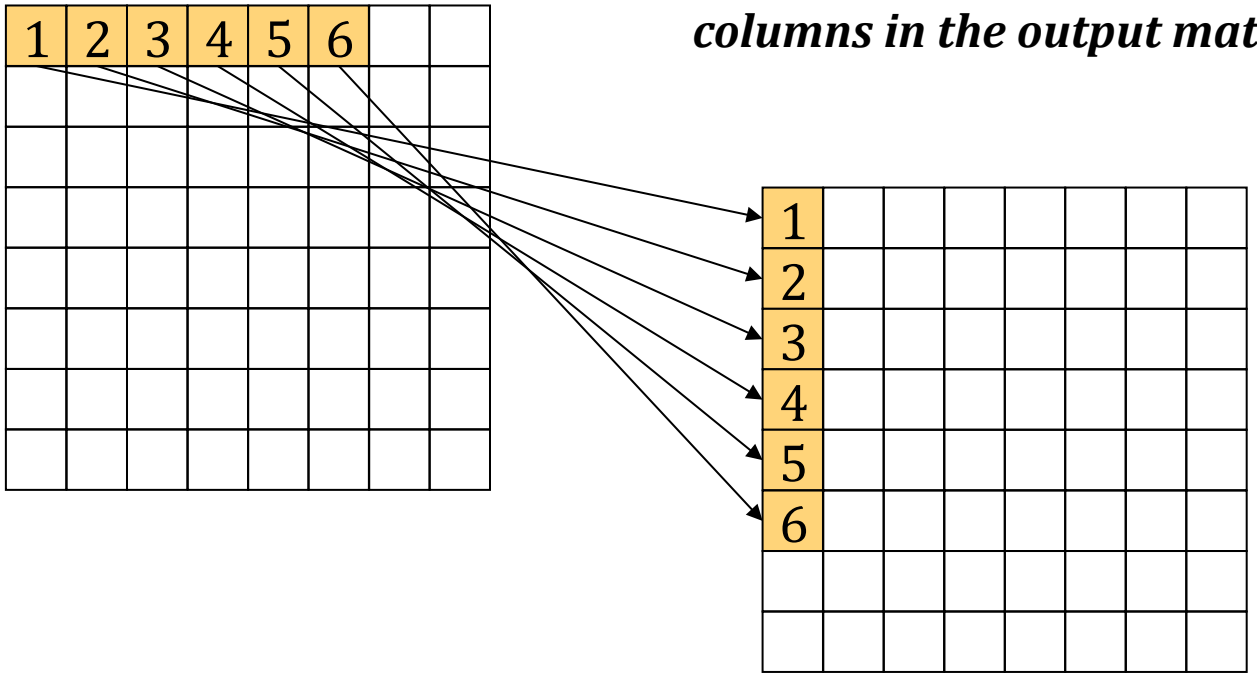
```
// Solve dA * dX = dB, where dA and dX are stored in GPU device memory.
// Internally, MAGMA uses a hybrid CPU + GPU algorithm.
void gpu_interface( magma_int_t n, magma_int_t nrhs )
{
    magmaDoubleComplex *dA=NULL, *dX=NULL;
    magma_int_t *ipiv=NULL;
    magma_int_t ldda = ((n+31)/32)*32;  // round up to multiple of 32 for best GPU performance
    magma_int_t lddx = ldda;
    magma_int_t info = 0;

    // magma malloc (GPU) routines are type-safe,
    // but you can use cudaMalloc if you prefer.
    magma_zmalloc( &dA, ldda*n );
    magma_zmalloc( &dX, lddx*nrhs );
    magma_imalloc_cpu( &ipiv, n );  // ipiv always on CPU
    if ( dA == NULL || dX == NULL || ipiv == NULL ) {
        fprintf( stderr, "malloc failed\n" );
        goto cleanup;
    }

    zfill_matrix_gpu( n, n, dA, ldda );
    zfill_rhs_gpu( n, nrhs, dX, lddx );

    magma_zgesv_gpu( n, 1, dA, ldda, ipiv, dX, ldda, &info );
    if ( info != 0 ) {
        fprintf( stderr, "magma_zgesv_gpu failed with info=%d\n", info );
    }
cleanup:
    magma_free( dA );
    magma_free( dX );
    magma_free_cpu( ipiv );
}
```

OPTIMIZATION CASE STUDY: MATRIX TRANSPOSE

Matrix transpose

- Write the rows of matrix A as columns of matrix A^T
- A bandwidth-limited problem
  - most of the time is spent moving data in memory rather than number crunching
- Utilizing the memory architecture effectively tends to be the biggest challenge for GPU algorithms
- Available as one of NVIDIA samples included with CUDA

Matrix transpose



Input rows are written as columns in the output matrix

The naïve matrix transpose

```
__global__ void transpose_naive(float *odata, float *idata, int width,int height)
{
    int xIndex, yIndex, index_in, index_out;

    xIndex = blockDim.x * blockIdx.x + threadIdx.x;
    yIndex = blockDim.y * blockIdx.y + threadIdx.y;

    if (xIndex < width && yIndex < height)
    {
        index_in = xIndex + width * yIndex;
        index_out = yIndex + height * xIndex;
        odata[index_out] = idata[index_in];
    }
}
```

Performance

- Transposing 8192x8192 matrix of SP floats, using monk M2070 GPUs, not counting memory transfers
- Naive implementation - **22.46** ms
- CUBLAS cublasSgeam implementation - **6.8** ms
- Naive implementation is 3.3 times slower
- Can we fix this?

Optimizing access to global memory

- A GPU has a large number of cores with great computational power, but they must be “fed” with data from global memory

- If too little computation done on core relative to memory transfer, then it becomes the bottleneck.
  - most of the time is spent moving data in memory rather than number crunching
  - for many problems this is unavoidable

- Utilizing the memory architecture effectively tends to be the biggest challenge in CUDA-fying algorithms

GPU memory is high bandwidth/high latency

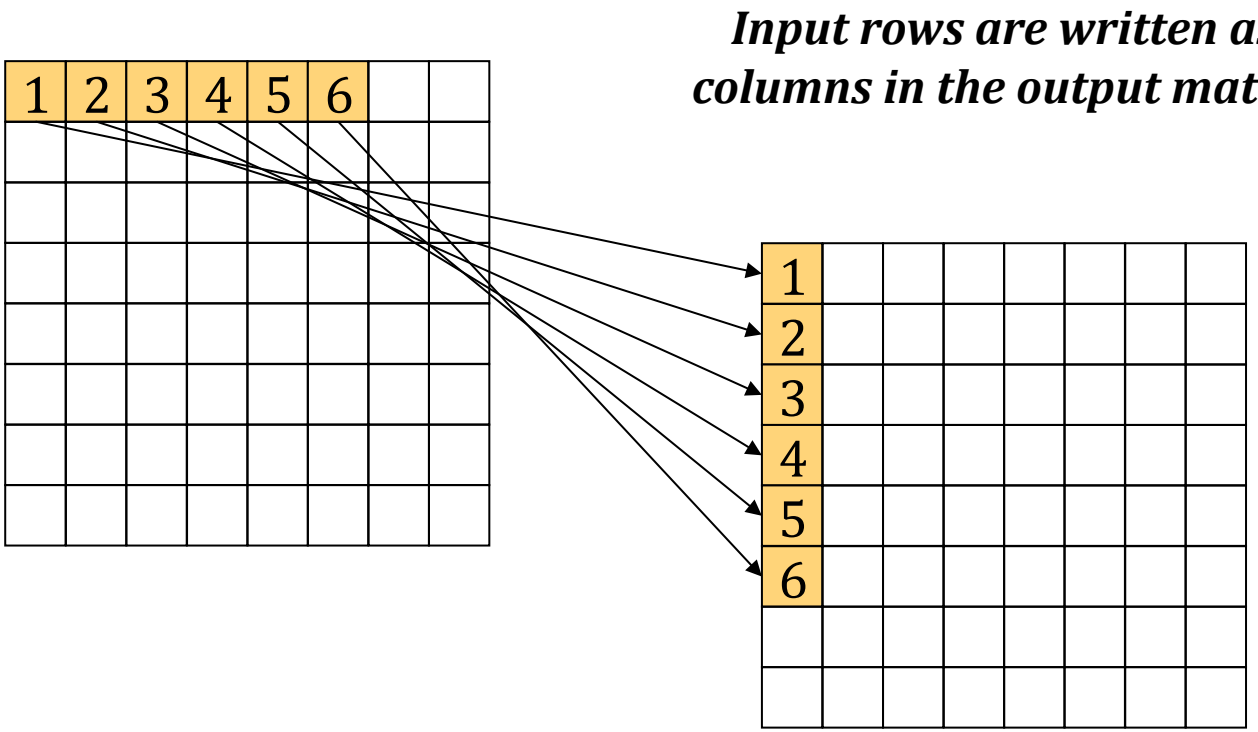
- A GPU has potentially high bandwidth for data transfer from global memory to cores. However, the latency for this transfer for any individual thread is also high (hundreds of cycles)
- Using many threads, latency can be overcome by hiding it among many threads.
  - group of threads requests some memory, while it is waiting for it to arrive, another group is computing
  - the more threads you have, the better this works
- The pattern of global memory access is also very important, as cache size of the GPU is very limited.

Global memory access is fast when coalesced

- It is best for adjacent threads belonging to the same warp (group of 32 threads) to be accessing locations adjacent in memory (or as close as possible)
- Good access pattern: thread i accesses global memory array member a[i]
- Inferior access pattern: thread i accesses global memory array member as a[i*nstride] where nstride >1
- Clearly, random access of memory is a particularly bad paradigm on the GPU

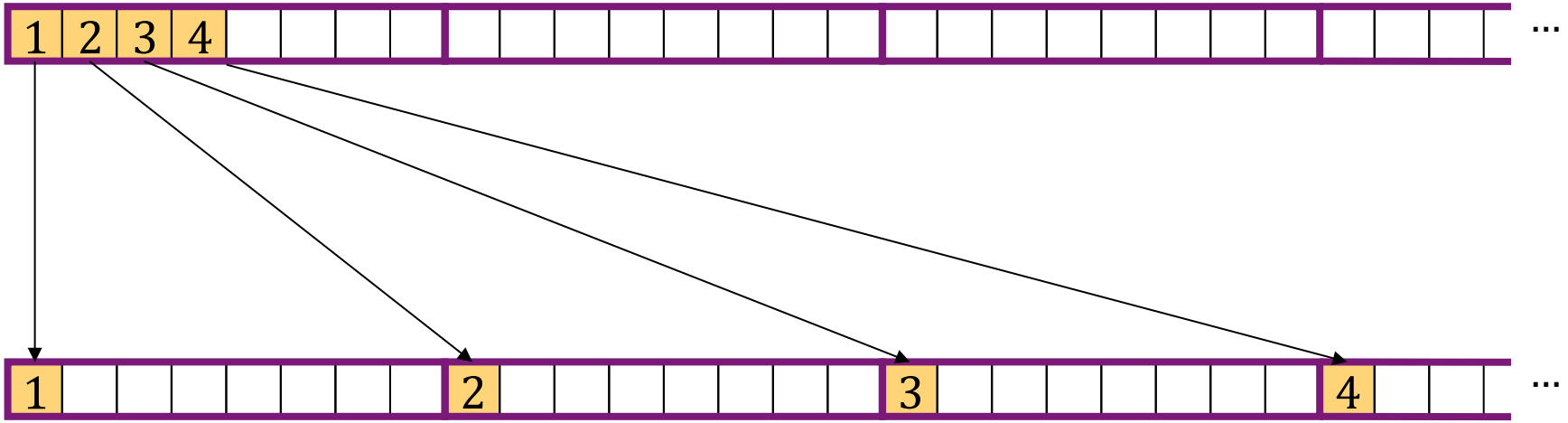
For some problems coalesced access is hard

- Example: matrix transpose
- A bandwidth-limited problem that is dominated by memory access



Naïve matrix transpose (cont.)

Since the matrices are stored as 1D arrays, here's what is actually happening:



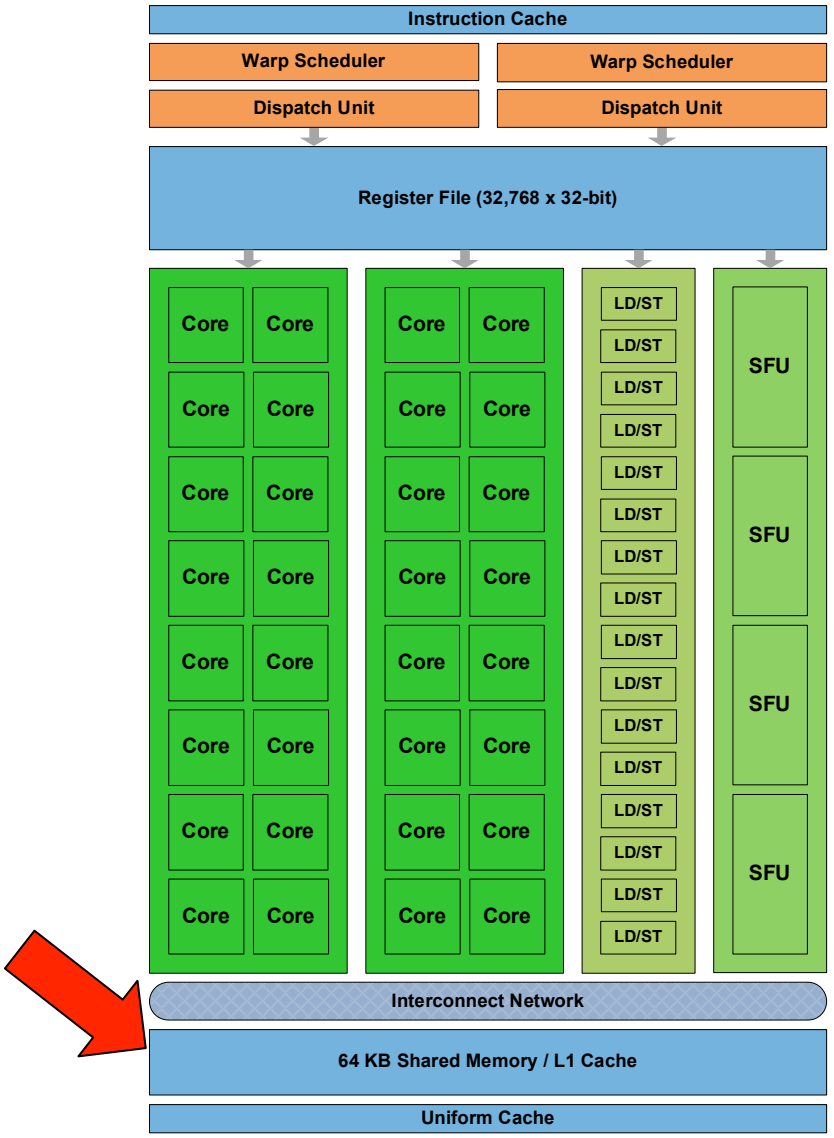
Can this problem be a avoided?

- Yes, by using a special memory which does not have a penalty when accessed in a non-coalesced way
- On the GPU this is the shared memory
- Shared memory accesses are faster than even coalesced global memory accesses. If accessing same data multiple times, try to put it in shared memory.
- Unfortunately, it is very small (48 KB or 16KB)
- Must be managed by the programmer

Shared memory

- Each multiprocessor has some fast on-chip shared memory
- Threads within a thread block can communicate using the shared memory
- Each thread in a thread block has R/W access to all of the shared memory allocated to a block
- Threads can synchronize using the intrinsic

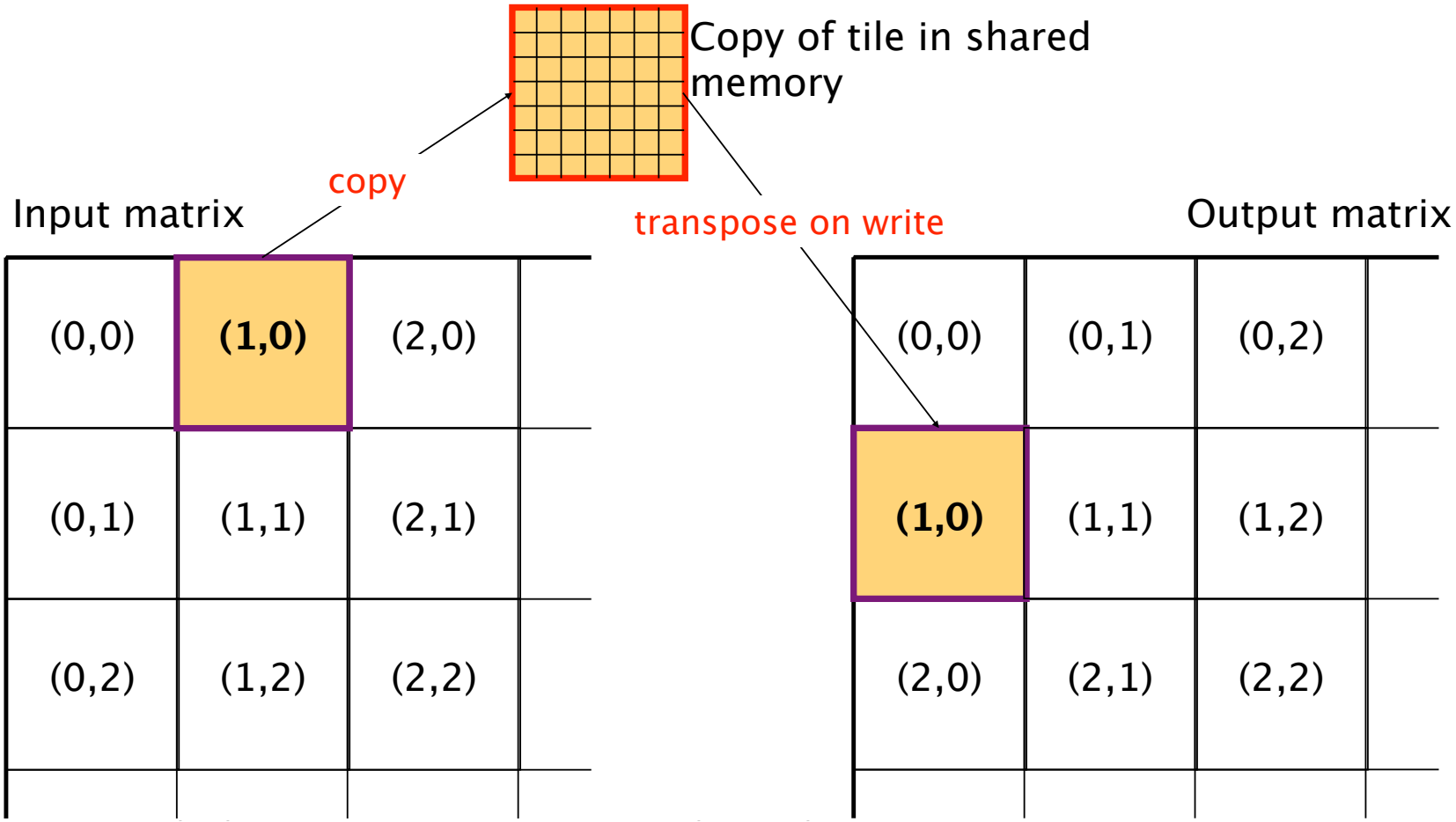
```
__syncthreads();
```



Using shared memory

- To coalesce the writes, we will partition the matrix into 32x32 tiles, each processed by a different thread block
- A thread block will temporarily stage its tile in shared memory by copying ti from the input matrix using coalesced reads
- Each tile is then transposed as it is written out to its properly location in the output matrix
- The main difference here is that the tile is written out using coalesced writes

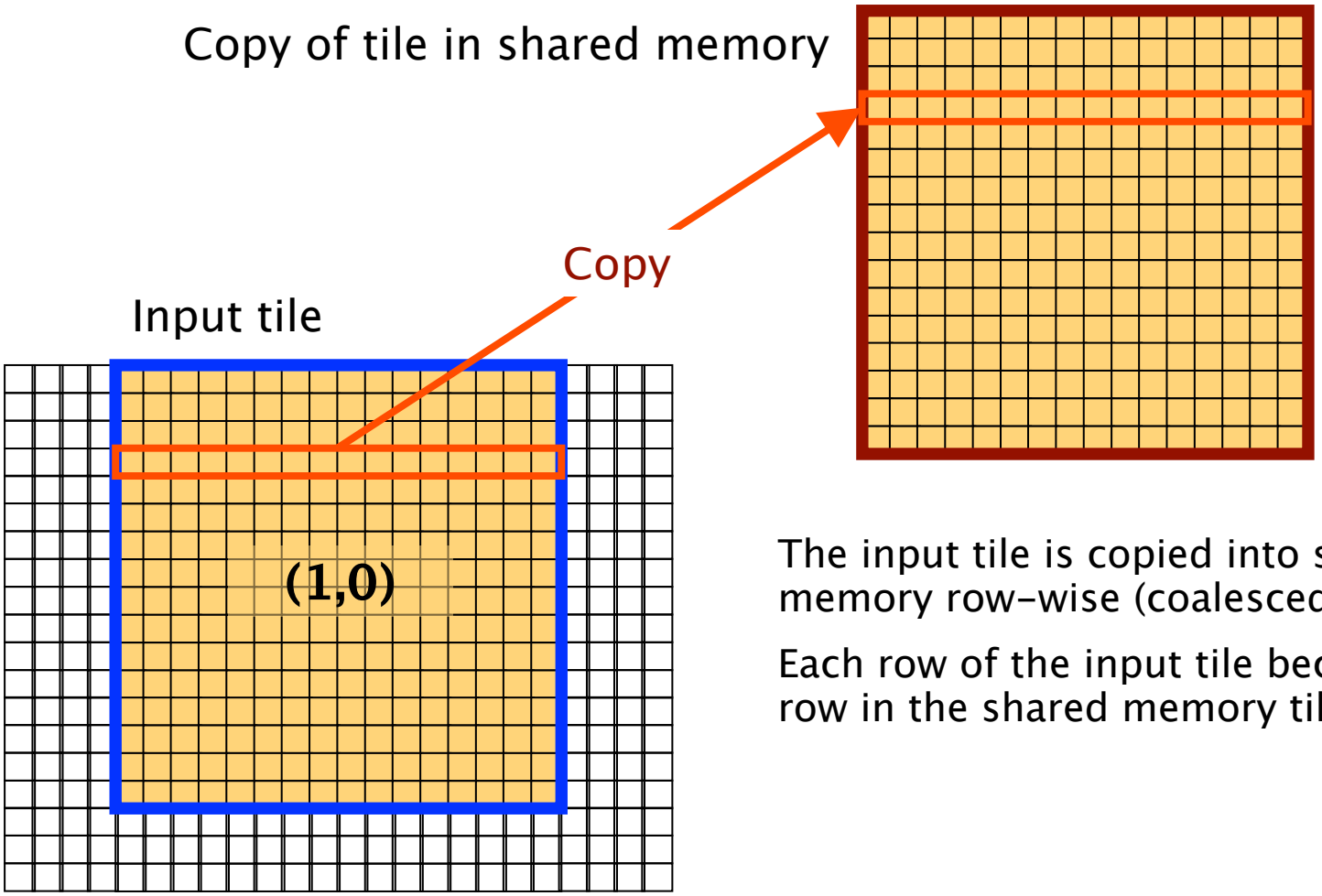
Optimized matrix transpose



Optimized matrix transpose (1)

```
__global__ void transpose(float *odata, float *idata,
                          int width, int height)
{
    __shared__ float block[BLOCK_DIM][BLOCK_DIM];
    unsigned int xIndex, yIndex, index_in, index_out;

    /* read the matrix tile into shared memory */
    xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
    if ((xIndex < width) && (yIndex < height))
    {
        index_in = yIndex * width + xIndex;
        block[threadIdx.y][threadIdx.x] = idata[index_in];
    }

    __syncthreads();

    /* write the transposed matrix tile to global memory */
    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
    if ((xIndex < height) && (yIndex < width))
    {
        index_out = yIndex * height + xIndex;
        odata[index_out] = block[threadIdx.x][threadIdx.y];
    }
}
```

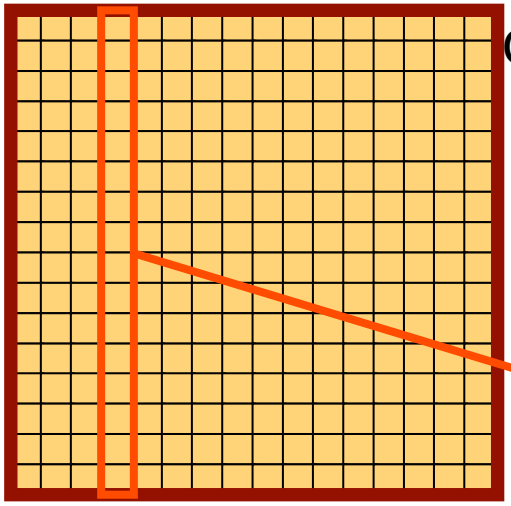
Optimized matrix transpose (cont.)



The input tile is copied into shared memory row-wise (coalesced reads).

Each row of the input tile becomes a row in the shared memory tile.

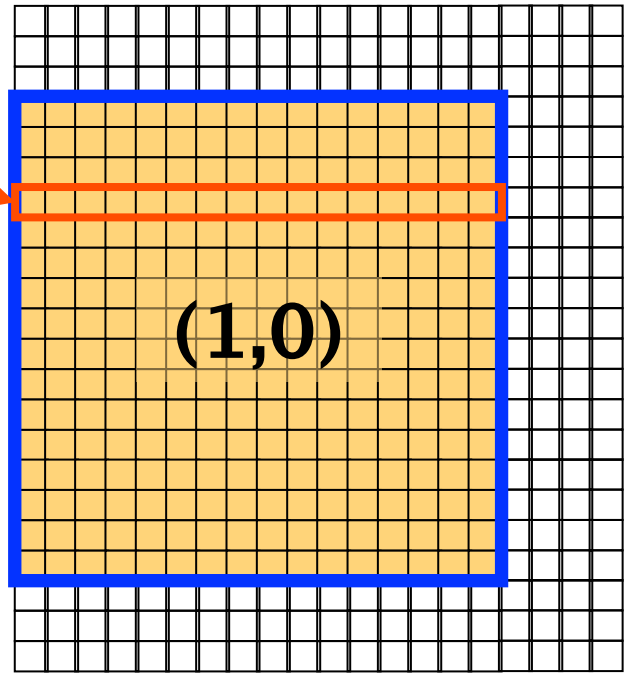
Optimized matrix transpose (cont.)



Copy of tile in shared memory

Transpose

Output tile



The output tile is written row-wise (coalesced writes) by copying the corresponding columns from shared memory.

One additional complication: bank conflicts

- Not a big concern but something to keep in mind
- Shared memory ***bank conflicts*** occur when the tile in shared memory is accessed column-wise
- Illustration of the need to really know the hardware when coding for GPU
- Bank conflicts matter only in highly optimised code where other sources of inefficiency have been eliminated

Shared memory banks

- To facilitate high memory bandwidth, the shared memory on each multiprocessor is organized into equally-sized ***banks*** which can be accessed simultaneously

- However, if more than one thread tries to access the same bank, the accesses must be serialized, causing delays
  - this situation is called a ***bank conflict***

- The banks are organized such that consecutive 32-bit words are assigned to consecutive banks

Shared memory banks (cont.)

- There are 32 banks, thus:

```
bank# = address % 32
```

- The number of shared memory banks is closely tied to the warp size
- Shared memory accesses are serviced such that the threads in the first half of a warp and the threads in the second half of the warp will not cause conflicts
- Thus we have `NUM_BANKS = WARP_SIZE`

Bank conflict solution

- In the matrix transpose example, bank conflicts occur when the shared memory is accessed column-wise as the tile is being written

- The threads in each warp access addresses which are offset from each other by `BLOCK_DIM` elements (with `BLOCK_DIM = 32`)

- Given 32 shared memory banks, that means that all accesses hit the same bank!

Bank conflict solution

- The solution is surprisingly simple – instead of allocating a `BLOCK_DIM x BLOCK_DIM` shared memory tile, we allocate a `BLOCK_DIM x (BLOCK_DIM+1)` tile

- The extra padding breaks the pattern and forces concurrent threads to access different banks of shared memory
  - the columns are no longer aligned on 32-word offsets
  - no additional changes to the device code are needed

Optimized matrix transpose (2)

```
 __global__ void transpose(float *odata, float *idata,
                           int width, int height)
{
    __shared__ float block[BLOCK_DIM][BLOCK_DIM + 1];
    unsigned int xIndex, yIndex, index_in, index_out;

    /* read the matrix tile into shared memory */
    xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
    if ((xIndex < width) && (yIndex < height))
    {
        index_in = yIndex * width + xIndex;
        block[threadIdx.y][threadIdx.x] = idata[index_in];
    }

    __syncthreads();

    /* write the transposed matrix tile to global memory */
    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
    if ((xIndex < height) && (yIndex < width))
    {
        index_out = yIndex * height + xIndex;
        odata[index_out] = block[threadIdx.x][threadIdx.y];
    }
}
```

Performance

- Tesla M2070 GPU on monk, transpose of 8192x8192 matrix of SP floats
- Averaged over 100 runs:

Size	time (ms)	Speedup
simple memory copy	8.7 ms	—
naive	22.5 ms	x 1.0
coalesced	20.8 ms	x 1.1
coalesced, bank optimized	11.1 ms	x 2.0
CUBLAS	6.8 ms	x 3.3

  - timings don't include data transfers!!!