# Parallel Programming

http://www.cs.bham.ac.uk/~hxt/2013/parallel-programming/

based on:

David B. Kirk and Wen-mei W. Hwu: Programming Massively Parallel Processors: A Hands-on Approach (second edition), 2013

and

Nvidia documentation

May 1, 2014

CUDA basics

2d arrays

Matrix multiplication

Coalescing

Prefix sum

Sparse matrix

OpenCL

Thrust

MPI

# Vector addition kernel and launch in CUDA C

```
__global__
void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

http://docs.nvidia.com/cuda/cuda-c-programming-guide/

# Vector addition kernel with blocks

http://docs.nvidia.com/cuda/cuda-c-programming-guide/#device-memory

```
// Device code
__global__ void VecAdd(float* A, float* B, float
    * C, int N)
{
     int i = blockDim.x * blockIdx.x + threadIdx.
         x;
     if (i < N)
          C[i] = A[i] + B[i];
}
```

```
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in
       host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...
```

```
// Allocate vectors in device memory
float* d_A;
cudaMalloc(&d_A, size);
float* d_B;
cudaMalloc(&d_B, size);
float* d_C;
cudaMalloc(&d_C, size);
```

```
// Copy vectors from host memory to device
    memory
cudaMemcpy(d_A, h_A, size,
    cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size,
    cudaMemcpyHostToDevice);

// Invoke kernel
...
```

```
    // Copy result from device memory to host
        memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size,
        cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

# CUDA memory management

```
cudaError_t cudaMalloc(void** devPtr,
                       size_t size)

cudaError_t cudaMemcpy(void* dst,
                       const void* src,
                       size_t count,
                       enum cudaMemcpyKind kind)
```

enumeration:

| | |
|---|---|
| cudaMemcpyHostToHost | Host -> Host |
| cudaMemcpyHostToDevice | Host -> Device |
| cudaMemcpyDeviceToHost | Device -> Host |
| cudaMemcpyDeviceToDevice | Device -> Device |
| cudaMemcpyDefault | |

Default based unified virtual address space

# xy coordinates

```
(0,0) (1,0) (2,0)
(0,1) (1,1) (2,1)
(0,2) (1,2) (2,2)
```

# yx coordinates

```
(0,0) (0,1) (0,2)
(1,0) (1,1) (1,2)
(2,0) (2,1) (2,2)
```

# 2D array in C layout example ("row major")

```
int a[2][3];
```

2-array of 3-array of ints

2D matrix of ints in memory:

```
    a[0][0]    a[0][1]    a[0][2]
    a[1][0]    a[1][1]    a[1][2]
```

```
a[i][j] = *(a + i * (3 * sizeof(int)) + j * sizeof(int))
```

```
a[1][1] = *(a + 1 * (3 * sizeof(int)) + 1 * sizeof(int))
          *(a + 4 * sizeof(int))
```

row index is scaled up (by 3)

# 2D array as 1D

```
int a[2][3];

2D:

    a[0][0]    a[0][1]    a[0][2]
    a[1][0]    a[1][1]    a[1][2]

1D:

a[0][0]  a[0][1]  a[0][2]  a[1][0]  a[1][1]  a[1][2]
```

# Matrix multiplication

$$C = A\,B$$

$$C_{i\,j} = \sum_{k=1}^{n} A_{i\,k}\,B_{k\,j}$$

$A$ is $p \times n$, $B$ is $n \times q$, then $C$ is $p \times q$
$i$ is row, scaled up by $n$ in memory
$j$ is column
For each dimension:
block index * block dim + thread index

# Matrix multiplication kernel in CUDA C, unoptimized

```
__global__
void gpuMM(float *A, float *B, float *C, int n)
{
    int row = blockIdx.y*blockDim.y+threadIdx.y;
    int col = blockIdx.x*blockDim.x+threadIdx.x;

    float sum = 0;
    for (int k = 0; k < n; k++) {
        sum += A[row * n + k]
             * B[k * n + col];
    }
    C[row * n + col] = sum;
}
```

See Kirk+Hwu, Figure 4.7 for code and Figure 4.6 for diagram, or

http://docs.nvidia.com/cuda/cuda-c-programming-guide/graphics/

matrix-multiplication-without-shared-memory.png

# Memory accesses in matrix mult

```
float sum = 0;
for (int k = 0; k < n; k++) {
    sum += A[row * n + k]
         * B[k * n + col];
}
C[row * n + col] = sum;
```

compute global memory access (CGMA) ratio:

2 float ops and 2 main memory accesses

Optimization: use locality for more "compute" per memory access

# Shared memory and barrier synchronization

More CUDA constructs:

```
__shared__ float[N];   // declaration

__syncthreads();       // statement
```

Nvidia says:
Shared memory is expected to be much faster than global memory. Any opportunity to replace global memory accesses by shared memory accesses should therefore be exploited.
`__syncthreads()` acts as a barrier at which all threads in the block must wait before any is allowed to proceed.
Shared memory and barrier synchronization are per thread block.

# Tiled matrix from Kirk+Hwu Fig 5.12, see also Fig 5.13

```
__global__
void MatrixMul(float* Md, float* Nd, float* Pd,
               int Width)
{
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;
    float Pvalue = 0;
```

# Tiled matrix part 2

```
    // Loop over the Md and Nd tiles required to
        compute the Pd element
    for (int m = 0; m < Width/TILE_WIDTH; m++) {
        // Collaborative loading of Md and Nd
            tiles into shared memory
        Mds[ty][tx] =
        Md[Row*Width + m*TILE_WIDTH + tx];
        Nds[ty][tx] =
        Nd[(m*TILE_WIDTH + ty)*Width + Col];
        __syncthreads();
        // inner loop inside tile
        for (int k = 0; k < TILE_WIDTH; k++)
            Pvalue += Mds[ty][k] * Nds[k][tx];
        __synchthreads();
    }
    Pd[Row*Width + Col] = Pvalue;
}
```

# Tiled has better compute/global memory access ratio

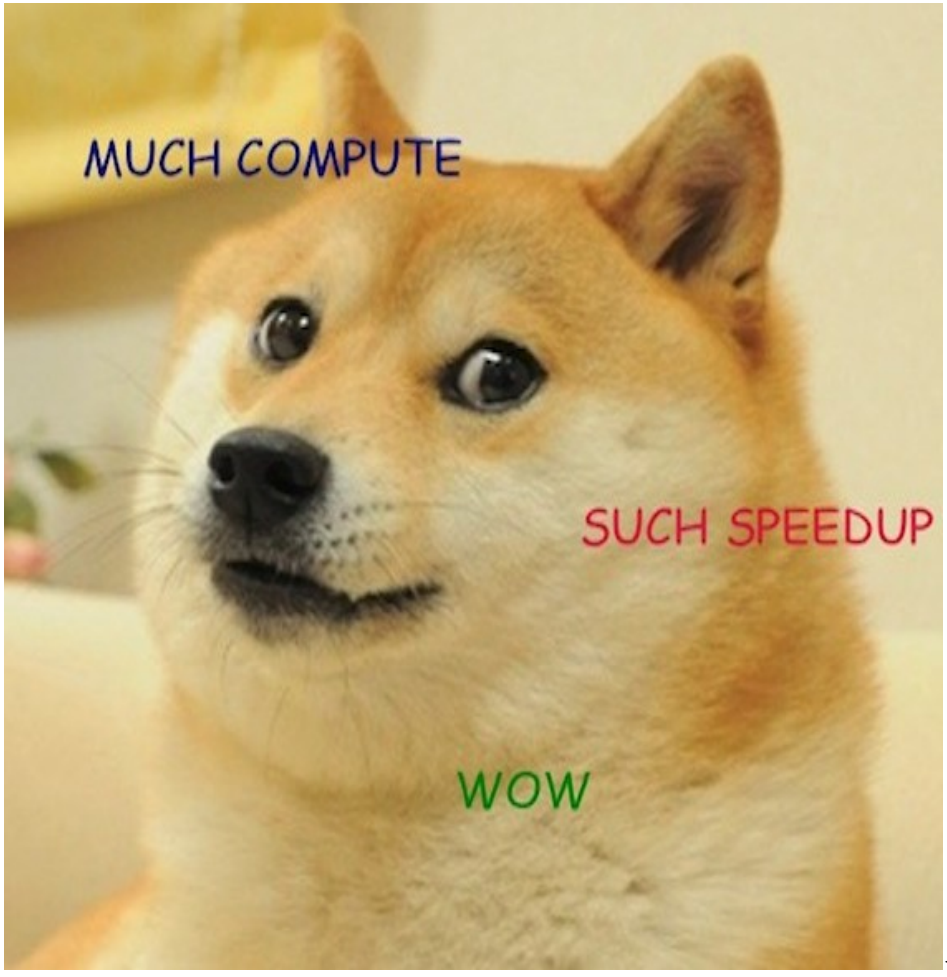
2 loads from global to shared,
then compute on shared for tile width steps

```
Mds[ty][tx] =
Md[Row*Width + m*TILE_WIDTH + tx];

Nds[ty][tx] =
Nd[(m*TILE_WIDTH + ty)*Width + Col];

for (int k = 0; k < TILE_WIDTH; k++)
     Pvalue += Mds[ty][k] * Nds[k][tx];
```

$\Rightarrow$ more "compute" per global memory access



←Doge meme

Pix or It Didn't Happen:
`http://docs.nvidia.com/cuda/cuda-c-programming-guide/graphics/matrix-multiplication-with-shared-memory.png`

# Coalesced memory access, see Fig 6.8

Example: Coalesced memory access in matrix mult
Good: each thread traverses matrix vertically through a column

```
for(int k = 0; ... ; k++) {
    ...
    N[k * Width + threadIdx.x]
}
```

$\Rightarrow$ all threads have the same `k` and successive `threadIdx.x`
$\Rightarrow$ memory accesses from different threads can be coalesced for the same k:

| | | | | |
|---|---|---|---|---|
| `k` | `N[k][0]` | `N[k][1]` | `N[k][2]` | `N[k][3]` |
| `k+1` | `N[k+1][0]` | `N[k+1][1]` | `N[k+2][2]` | `N[k+3][3]` |

# Control divergence

Control (`if`, `while`) may depend on thread index ⇒ less SIMD
Example: control divergence in reduction kernel
Only every 2nd, 4th, 8th, ... kernel takes else branch

```
if (threadIdx.x % s) ...
```

Fig 6.3 vs Fig 6.5: active kernels compact on one side,
not scattered over different thread blocks
⇒ less (no) control divergence
For reduce operations, see
`http://www.cs.cmu.edu/~guyb/papers/Ble93.pdf`

# Array reversal, unoptimized - Lab 1

```
// step 3: implement the kernel
__global__
void reverseArrayBlock(int *d_b, int *d_a) {
  // source index
  int src = threadIdx.x;
  // destination index (reversed)
  int dst = blockDim.x - threadIdx.x - 1;
  // no swap, just overwrite the destination
  d_b[dst] = d_a[src];
}
```

# Reduction (Blelloch 1993)

Definition (Prefix sum) Let $\oplus$ be a binary operator on a set $A$. The *reduction* for $\oplus$ of a sequence

$$a_0, a_1, \ldots, a_{n-1}$$

of elements of $A$ is defined as the element of $A$ given by

$$a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1}$$

# Monoid

Let $A$ be a set. Let $\oplus$ be a binary operation on $A$.
$\oplus$ is associative if for all $a$, $b$, $c$ in $A$,

$$a \oplus (b \oplus c) \quad = \quad (a \oplus b) \oplus c$$

Let $I$ be an element of $A$. $I$ is the identity (or unit) if for all $a$ in $A$,

$$a \oplus I = a \text{ and } I \oplus a = a$$

If $A$ has an associate operation $\oplus$ and an identity $I$, it is called a *monoid*.

# Prefix sum (Blelloch 1993)

Definition (Prefix sum) Let $\oplus$ be a binary operator on a set $A$. The *prefix sum* for $\oplus$ of a sequence

$$a_0, a_1, \ldots, a_{n-1}$$

of elements of $A$ is defined as the sequence

$$p_0, p_1, \ldots, p_{n-1}$$

of elements of $A$ where

$$\begin{aligned} p_0 &= a_0 \\ p_{j+1} &= p_j \oplus a_{j+1} \end{aligned}$$

Informally,

$$a_0, (a_0 \oplus a_1), (a_0 \oplus a_1 \oplus a_2), \ldots, (a_0 \oplus \ldots \oplus a_{n-1})$$

# Prescan (Blelloch 1993)

Definition (Prescan) Let $\oplus$ be a binary operator on a set $A$ and $I$ an element of $A$. The *prescan* for $\oplus$ and $I$ of a sequence

$$a_0\, a_1 \ldots a_{n-1}$$

of elements of $A$ is defined as the sequence

$$p_0\, p_1 \ldots p_{n-1}$$

of elements of $A$ where

$$\begin{aligned} p_0 &= I \\ p_{j+1} &= p_j \oplus a_j \end{aligned}$$

Informally,

$$I, (I \oplus a_0), (I \oplus a_0 \oplus a_1), \ldots, (I \oplus a_0 \oplus \ldots \oplus a_{n-2})$$

# Prefix sum and scan (Blelloch 1993)

Theorem. The prescan for a monoid on an input sequence of length $n$ can be compute in $O(\log n)$ times on a Parallel Random Access Memory machine.
Proof: via Blelloch's upsweep and downsweep algorithms. The trees are of $\log_2 n$ height. The tree transformations are correct because $\oplus$ is associative.

# Example: upsweep with * in $3 = \log_2 8$ steps

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 2 | 3 | 2 | 1 | 2 | 2 | 1 |
| 2 | 4 | 3 | 6 | 1 | 2 | 2 | 2 |
| 2 | 4 | 3 | 24 | 1 | 2 | 2 | 4 |
| 2 | 4 | 3 | 24 | 1 | 2 | 2 | 96 |

# Example: downsweep with * and 1 in $3 = \log_2 8$ steps

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 4 | 3 | 24 | 1 | 2 | 2 | 1 |
| 2 | 4 | 3 | 1 | 1 | 2 | 2 | 24 |
| 2 | 1 | 3 | 4 | 1 | 24 | 2 | 48 |
| 1 | 2 | 4 | 12 | 24 | 24 | 48 | 96 |

This is indeed the prescan for * and 1 of the original sequence:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 2 | 3 | 2 | 1 | 2 | 2 | 1 |

So log. Wow.

# Sparse matrices, history and motivation

Mathematical finance emerging in the 1950s, Harry Markowitz:

> *He tried to predict this with a square matrix where each row and column represented an industry.*
>
> *It struck me that our matrices were mostly full of zeros.*
>
> *Then I thought, well, maybe we could get the computer to do the same thing. This led to Sparse Matrices.*

Further reading:
Markowitz interview: `http://conservancy.umn.edu/bitstream/11299/107467/1/oh333hmm.pdf`

Robert Shiller course: `https://class.coursera.org/financialmarkets-001`

# COO: coordinate format

Three arrays:

1. non-zero elements
2. column indices for each non-zero element
3. row indices for each non-zero element

Example:

```
3    0    1    0
0    0    0    0
0    2    4    1
1    0    0    1
```

In COO format:

# COO: coordinate format

Three arrays:

1. non-zero elements
2. column indices for each non-zero element
3. row indices for each non-zero element

Example:

```
3    0    1    0
0    0    0    0
0    2    4    1
1    0    0    1
```

In COO format:

```
[3, 1, 2, 4, 1, 1, 1]
```

# COO: coordinate format

Three arrays:

1. non-zero elements
2. column indices for each non-zero element
3. row indices for each non-zero element

Example:

```
3    0    1    0
0    0    0    0
0    2    4    1
1    0    0    1
```

In COO format:

```
[3, 1, 2, 4, 1, 1, 1]
```

```
[0, 2, 1, 2, 3, 0, 1]
```

# COO: coordinate format

Three arrays:

1. non-zero elements
2. column indices for each non-zero element
3. row indices for each non-zero element

Example:

```
3    0    1    0
0    0    0    0
0    2    4    1
1    0    0    1
```

In COO format:

```
[3, 1, 2, 4, 1, 1, 1]

[0, 2, 1, 2, 3, 0, 1]

[0, 0, 2, 2, 2, 3, 3]
```

# CSR: compressed sparse row

Three arrays:

1. non-zero elements
2. column indices for each non-zero element
3. row pointers, beginning and end of an interval for each row

Example:

```
3    0    1    0
0    0    0    0
0    2    4    1
1    0    0    1
```

In CSR format:

```
[3, 1, 2, 4, 1, 1, 1]

[0, 2, 1, 2, 3, 0, 3]

[0, 2, 2, 5, 7]
```

# CSR: pointers encode rows intervals between two pointers

```
3   0   1   0
0   0   0   0
0   2   4   1
1   0   0   1
```

In CSR format:

```
[3, 1, 2, 4, 1, 1, 1]

[0, 2, 1, 2, 3, 0, 3]

[0, 2, 2, 5, 7]
```

Encoding of rows as closed/open intervals between two pointers:

```
[0, 2[      row 0
[2, 2[      row 1
[2, 5[      row 2
[5, 7[      row 3
```

# SpMV, spare matrix times vector on GPU

SpMV terminology: SpM= spare matrix, V =vector

$$y = A \cdot x$$

- row = thread index
- loop over row pointers for interval of current row
- index into first array to get matrix element
- index into second array to find column position in y

# SpMV, spare matrix times vector kernel using CSR

1. `data` = non-zero elements of A
2. `col_index` = column indices for each non-zero element of A
3. `row_ptr` = row interval pointers of A
4. `x` = vector to be multiplied with A

```
int row = blockIdx.x * blockDim.x + threadIdx.x;

float sum = 0;

int row_start = row_ptr[row];
int row_end = row_ptr[row + 1];

for (int i = row_start; i < row_end; i++)
      sum += data[i]
           * x[col_index[i]];

y[row] = sum;
```

# SpMV, spare matrix times vector kernel using CSR

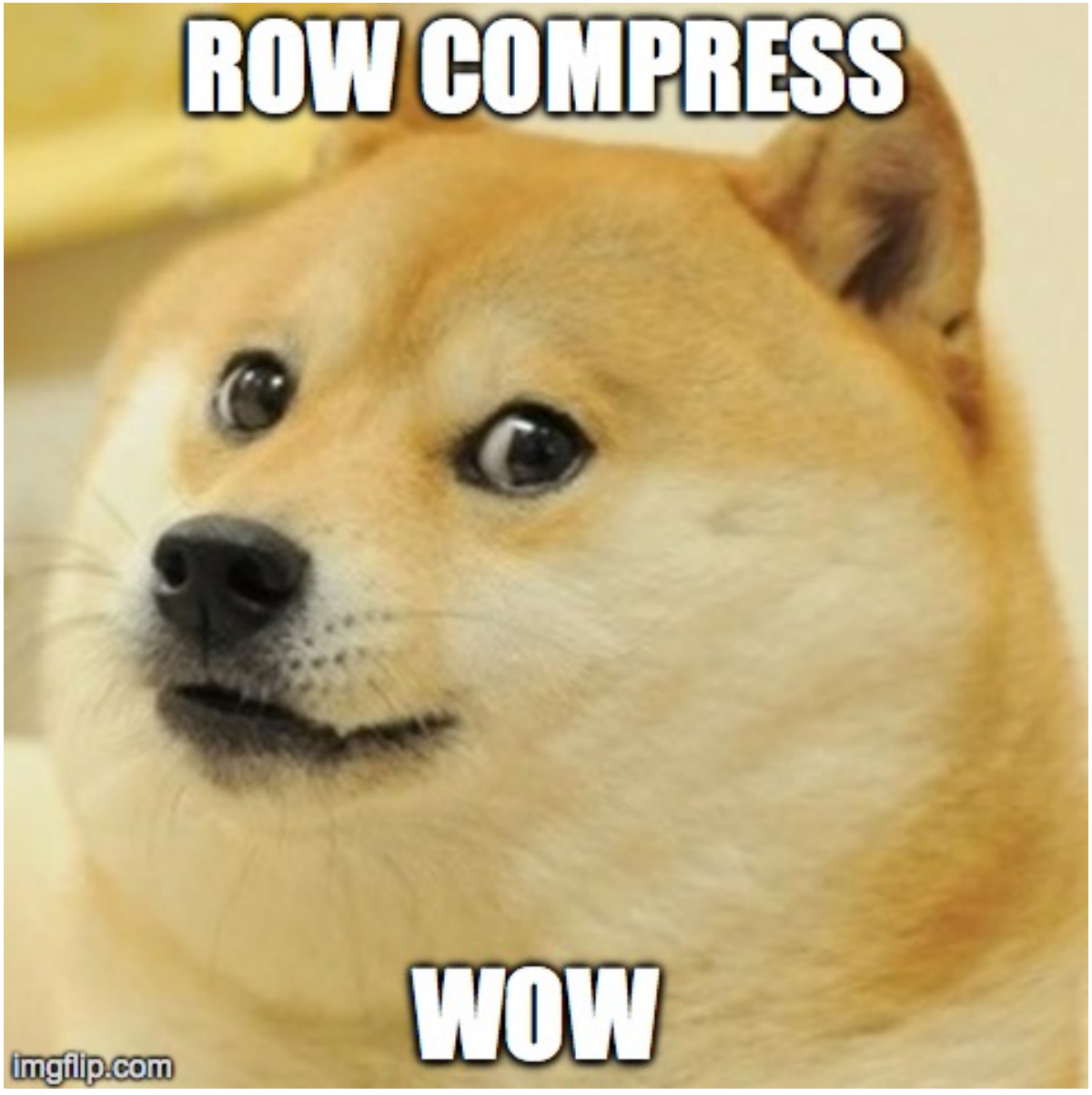
Disadvantages of CSR matrix multiplication:

- non-coalesced accesses
  scattering via array in index
  `x[col_index[i]]`
  Also bad for cache
- control flow divergence due to different loop lengths
  lengths come from array

```
int row_start = row_ptr[row];
int row_end = row_ptr[row + 1];

for (int i = row_start; i < row_end; i++)
```

# Before



# After

# ELL format: pad out with zeros to make rows uniform

```
3    0    1    0
0    0    0    0
0    2    4    1      longest row
1    0    0    1
```

becomes

```
3    1    0
0    0    0
2    4    1
1    1    0
```

with column indices, also padded with 0s:

```
0    2    0
0    0    0
1    2    3
0    3    0
```

# ELL format: traversal by threads

```
3   1   0   thread 0
0   0   0   thread 1
2   4   1   thread 2
1   1   0   thread 3
```

Column indices:

```
0   2   0
0   0   0
1   2   3
0   3   0
```

Column-major in memory:

```
3   0   2   1   1   0   4   1   0   0   1   0
0   0   1   0   2   0   2   3   0   0   3   0
                |                   |
```

# ELL SpMV kernel

ELL is less compact but more regular than CSR.
`num_elem` = elements per row with padding

```
for (int i = 0; i < num_elem; i++) {
      sum += data[row + i * num_rows]
           * x[col_index[row + i * num_rows]];
}
y[row] = sum;
```

Note: no control divergence

```
for (int i = 0; i < num_elem; i++)
```

vertical traversal of matrix in each thread:

```
row + i * num_rows
```

Stride is `num_rows`

# COO revisited

COO format has 3 arrays:

1. non-zero elements
2. column indices for each non-zero element
3. row indices for each non-zero element

Equivalently, a set of

$$(x, i, j)$$

COO allows reordering

# Hybrid HYB

Use COO for non-sparse rows

CSR or ELL on remaining rows

Host may perform COO computations

Further refinement: Jagged diagonal storage JDS

Sort rows by length

# Summary

Sparse matrix
COO, CSR, ELL formats
SpMV kernels:

$$y = A \cdot x$$

or

$$y = A \cdot x + y$$

Loops in kernels use the arrays to index into matrix and vector
Trade-off between compactness and regularity
Hard to get CGMA better than 1
Compare: dense matrix, data sharing and tiling

# Further reading

We have followed Kirk+Hwu Chapter 10.
For experimental data and examples of sparse matrices, see "Efficient Sparse Matrix-Vector Multiplication on CUDA" by Nathan Bell and Michael Garland
`http://www.nvidia.com/docs/IO/66889/nvr-2008-004.pdf`

# OpenCL

See Kirk+Hwu Sections 14.1 and 14.2.
OpenCL code is broadly similar to CUDA.

Matrix times vector in OpenCL:
`https://developer.nvidia.com/opencl#oclMatVecMul`

Parallel reduction in OpenCL:
`https://developer.nvidia.com/opencl#oclReduction`

# Thrust: C++ templates for GPU programming

Introduction to Thrust:
`http://thrust.googlecode.com/files/An%20Introduction%20To%20Thrust.pdf`

Thrust resources:
`https://code.google.com/p/thrust/`

Kirk+Hwu: Chapter 16

Background on C++ templates:
`http://www.cs.bham.ac.uk/~hxt/2013/c-programming-language/intro-to-templates.pdf`

# C++ in the Thrust code examples

<...> is C++ template instantiation

C++ local variable declaration:

```
thrust::host_vector<int> h_vec(1 << 24);
```

Operator overloading: = and [ ] for Thrust vectors

We can use Thrust vectors much as if they were C arrays.

The details of the kernels and GPU memory are hidden most of the time.

# C++ templates

- C++ templates are enormously complicated
- templates are important: Standard Template Library
- templates interact/collide with other C++ features
- templates push the C++ type system to breaking point
- templates allow compile-time computation $\Rightarrow$ zero overhead
- templates are a different language design dimension from object-orientation
- one way to use them is similar to polymorphism in functional languages
- In this module, we will build on your knowledge of functional programming

# Templates and polymorphism

There are two kinds of templates in C++:

1. Class templates
2. Function templates

These correspond roughly to polymorphic data types and polymorphic functions in functional languages.

# Polymorphism in functional languages

```
# [1; 2; 3];;
- : int list = [1; 2; 3]

type 'a bt = Leaf of 'a
           | Internal of 'a bt * 'a bt;;

# let twice f x = f(f x);;
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
```

# Templates: keyword `template`

```
template<typename T>
struct s {
 ... T ... T ...
 };
```

Then instantiating the template with argument `A` in `s<A>` is like

```
struct sA {
 ... A ... A ...
 };
```

Compare: $\lambda$ calculus.

# Templates: type parameter

```
template<typename T>
struct S
{
    // members here may depend on type parameter
       T
    T data;             // for example a data member
    void f(T);          // or a member function
    using t = T;        // or making t an alias for
       T
};
```

# Class template example

```
template<typename T>
struct Linked
{
    T head;
    Linked<T>* tail;
};
```

Class template - other keywords

```
template<class T>
class Linked
public:
{
    T head;
    Linked<T>* tail;
};
```

# Telling a template what to do

Suppose we have a Thrust template for transforming vectors.
We may want to give it a function that returns

$$a * x + y$$

"saxpy": set to a times x plus y
We could use a C function:

```
int saxpy(int x, int y)
{
    return a * x + y;
}
```

Limitation: a fixed a is hardwired into the function.

# Functors/function objects in C++

```
class cadd {
private:
    int n;
public:
    cadd(int n) { this->n = n; }

    int operator() (int m) { return n + m; }
};

int main(int argc, const char * argv[])
{
    cadd addfive(5);

    std::cout << addfive(7);
}
```

This prints 12.

# In OCAML

```
# let cadd n m = n + m;;
val cadd : int -> int -> int = <fun>
# cadd 5 7;;
- : int = 12

# List.map (cadd 5) [1; 2; 3];;
- : int list = [6; 7; 8]
```

This is similar to programming in Thrust with a transformer over a vector and a functor.

# Functor as template argument

```
template <typename T, typename Op>
T twice(T x, Op f)
{
    return f(f(x));
}

int main(int argc, const char * argv[])
{
    cadd addfive(5); // create function object

    cout << twice<int, cadd>(10, addfive) <<
        endl;
    cout << twice(10, addfive) << endl;
}
```

This prints 20 twice.

# Functors in Thrust - Kirk+Hwu Figure 16.4

```
struct saxpy_functor
{
    const float a;

    saxpy_functor(float _a) : a(_a) {}

    __host__ __device__
        float operator()(const float& x, const
            float& y) const
        {
            return a * x + y;
        }
};
```

This functor can be used by

`thrust::transform`

# MPI = Message Passing Interface

MPI stands for Message Passing Interface
specification, independent of implementation
can be called from C or FORTRAN, OCAML, ...

point-to-point send and receive
broadcast and barriers

MPI is unlike CUDA:

- distributed, multiple hosts in network
- message passing $\neq$ shared memory

# MPI and CUDA

In some ways, MPI is like CUDA:

- SPMD
- MPI rank = CUDA id
- branch on id $\Rightarrow$ different control paths through same program
- barrier synchronisation

# MPI example code

```
int main(int argc, char *argv[])
{
    int myrank, message_size=50, tag=99;
    char message[message_size];
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    if (myrank == 0) {
        MPI_Recv(message, message_size, MPI_CHAR
            , 1, tag, MPI_COMM_WORLD, &status);
        printf("received \"%s\"\n", message);
    }
    else {
        strcpy(message, "Hello, there");
        MPI_Send(message, strlen(message)+1,
            MPI_CHAR, 0, tag, MPI_COMM_WORLD);
    }
    MPI_Finalize();
}
```

# MPI communication

point to point: send and receive:

```
MPI_Send(buf,count,datatype,dest,tag,comm,ierr)

MPI_Recv(buf,count,datatype,source,tag,comm,
    status,ierr)
```

One process sends to everybody:

```
MPI_Bcast(buffer,count,datatype,root,comm,ierr)
```

All processes send to root process and the operation op is applied

```
MPI_Reduce(sendbuf,recvbuf,count,datatype,op,
    root,comm,ierr)
```

Synchronize all processes

```
MPI_Barrier(comm,ierr)
```

# More MPI functions

All processes send a different piece of data to one single root process which gathers everything (messages ordered by index)

```
MPI_Gather(sendbuf,sendcnt,sendtype,recvbuf,
    recvcount, recvtype,root,comm,ierr)
```

All processes gather everybody elses pieces of data

```
MPI_Allgather(sendbuf,sendcount,sendtype,recvbuf
    ,recvcount,recvtype,comm,info)
```

One root process send a different piece of the data to each one of the other processes

```
MPI_Scatter(sendbuf,sendcnt,sendtype,recvbuf,
    recvcnt, recvtype,root,comm,ierr)
```

Each process performs a scatter operation, sending a distinct message to all the processes in the group in order by index.

```
MPI_Alltoall(sendbuf,sendcount,sendtype,recvbuf,
    recvcnt, recvtype,comm,ierr)
```

# CUDA and MPI

pinned/page-locked versus pageable memory
`malloc` gives pageable memory

```
cudaError_t cudaHostAlloc(void **pHost,
                          size_t size,
                          unsigned int flags)
```

Allocates size bytes of host memory that is page-locked