CS 470 Spring 2019

Mike Lam, Professor

Parallel Languages & CUDA

Graphics and content taken from the following:

http://dl.acm.org/citation.cfm?id=2716320
http://chapel.cray.com/papers/BriefOverviewChapel.pdf
http://arxiv.org/pdf/1411.1607v4.pdf
https://en.wikipedia.org/wiki/Cilk

Parallel languages

- Writing efficient parallel code is hard
- We've covered two generic paradigms ...
  - Shared-memory
  - Distributed message-passing
- … and three specific technologies (but all in C!)
  - Pthreads
  - OpenMP
  - MPI
- Can we make parallelism easier by changing our language?
  - Similarly: Can we improve programmer *productivity*?

Productivity

- Economic definition:

$$Productivity = \frac{Output}{Input}$$

- What does this mean for parallel programming?
  - How do you measure *input*?
    - Bad idea: size of programming team
    - "The Mythical Man Month" by Frederick Brooks
  - How do you measure *output*?
    - Bad idea: lines of code

Productivity vs. Performance

- General idea: Produce **better** code **faster**
  - **Better** can mean a variety of things: speed, robustness, etc.
  - **Faster** generally means time/personnel investment

- Problem: productivity often trades off with performance
  - E.g., Python vs. C or Matlab vs. Fortran
  - E.g., garbage collection or thread management

Why?

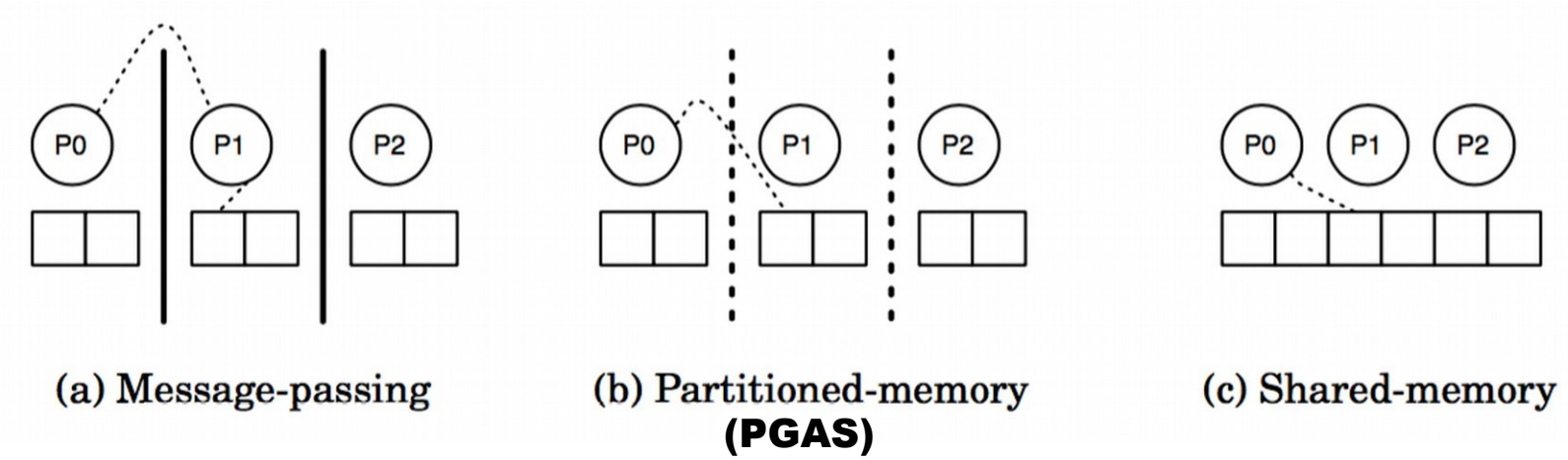
Complexity

- Core issue: handling **complexity**
- Tradeoff: developer effort vs. system effort
  - Hiding complexity from the developer increases the complexity of the system
  - Higher burden on compiler and runtime systems
  - Implicit features cause unpredictable interactions
  - More middleware increases chance of interference and software regressions
  - In distributed systems: locality matters **a lot**, but is difficult to automate

Holy Grail

PGAS

- Partitioned Global Address Space (PGAS)
  - Hybrid of distributed message-passing and shared-memory
  - Programmer sees one global address space
    - Each process has its own virtual address space “under the hood”
  - Compiler/runtime must sort out the communication
    - Often using a message-passing library like MPI or GASnet “under the hood”



Parallel Languages (Mostly PGAS)

- Erlang [Ericsson, 1986], Haskell [1990], and Clojure [2007]
  - Functional languages; most include explicit or implicit parallelism
- High Performance Fortran (HPF) [1993]
  - Designed by committee
- Academic languages
  - ZPL [UW, 1994]
  - Cilk [MIT, 1994] and Cilk Plus [Intel, 2010]
  - Titanium [UC Berkeley, 1998]
- Coarray Fortran (CAF) [1998]
  - Now officially part of the Fortran 2008 standard
- Unified Parallel C (UPC) [1999]
- HPCS languages [starting 2002]
- Julia [2012]

High-Performance Fortran

- Motivation: higher abstractions for parallelism
  - Predefined data distributions and parallel loops
  - Optional **directives** for parallelism (similar to OpenMP)
- Development based on Fortran 90
  - Proposed 1991 w/ intense design efforts in early 1990s
  - Wide variety of influences on the design committee
  - Standardized in 1993 and presented at Supercomputing '93

```
1  REAL A(1000,1000), B(1000,1000)
2  !HPF$ DISTRIBUTE A(BLOCK,*)
3  !HPF$ ALIGN B(I,J) WITH A(I,J)
4  DO J = 2, N
5    DO I = 2, N
6      A(I,J)=(A(I,J+1)+2*A(I,J)+A(I,J-1))*0.25 &
7           + (B(I+1,J)+2*B(I,J)+B(I-1,J))*0.25
```

Listing 8: Simple relaxation loop in HPF.

For the full story, see "The Rise and Fall of High Performance Fortran: An Historical Object Lesson"
`http://dl.acm.org/citation.cfm?id=1238851`

High-Performance Fortran

- Issues
  - Immature compilers and no reference implementation
  - Poor support for non-standard data distributions
  - Poor code performance; difficult to optimize and tune
  - Slow uptake among the HPC community
- Legacy
  - Effort in 1995-1996 to fix problems with HPF 2.0 standard
  - Eventually dropped in popularity and was largely abandoned
  - Some ideas still had a profound influence on later efforts

ZPL ("Z-level Programming Language")

- Array programming language (UW, 1994)
  - All parallelism is implicit
  - Regular data structures with grid alignments
  - Explicit regions and directions

TW = "the world"
NN = "num of neighbors"

```
1  program Life;
2    config const n : integer
3    region R = [1..n, 1..n];
4    direction nw = [-1, -1]; no = [-1, 0]; ne = [-1, 1];
5              w  = [ 0, -1];                e  = [ 0, 1];
6              sw = [ 1, -1]; so = [ 1, 0]; se = [ 1, 1];
7  var  TW : [R] boolean;
8       NN : [R] sbyte;
9  procedure Life();
10 begin -- Initialize the world
11   [R] repeat
12           NN := TW@^nw + TW@^no + TW@^ne
13               + TW@^w  +          TW^@e
14               + TW@^sw + TW@^so + TW@^se;
15           TW := (TW & NN = 2) | (NN = 3)
16       until !(|<< TW);
17 end;
```

Listing 9: Conway's Game of Life in ZPL.

field of cells

@^ "wrap-at" operator (shifts array/matrix)

region specifier

<< reduction operator (applied here on boolean OR)

Directly influenced the Chapel language

Cilk and Cilk Plus

- Extension to C language, not PGAS (MIT, 1994)
  - New `spawn` and `sync` keywords for fork-join task parallelism
    - Similar to Pthreads or OpenMP tasks
  - New `cilk_for` construct for parallel loops
    - Similar to OpenMP parallel for loops
  - Acquired by Intel in 2009, deprecated in 2017

```
cilk int fib(int n)
{
    if (n < 2)
        return n;
    else {
       int x, y;
       x = spawn fib(n - 1);
       y = spawn fib(n - 2);
       sync;
       return x + y;
    }
}
```

```
void loop(int *a, int n)
{
    cilk_for (int i = 0; i < n; i++) {
        a[i] = f(a[i]);
    }
}
```

Co-Array Fortran (CAF) [1998]

Extension to Fortran

```
1  INTEGER n
2  ...
3  n = 5
```

(a) Allocate private integer.

```
1  INTEGER n[*]
2  ...
3  n[p] = 5
```

co-array declared with [*]

(b) Allocate shared integer by creating a co-array.

Fig. 7: Both code fragments allocate one integer n for each place.

```
1  ! global_sum
2  INTEGER :: x(n)[*]                    ! array with a co-array
3  INTEGER :: local_temp(n)              ! array without a co-array
4  INTEGER :: me, mypartner              ! indices of places
5  INTEGER :: n, bit, i, iterations      ! other variables
6
7  iterations = log2_images()
8  bit = 1
9  me = this_image(x)
10 DO i = 1,iterations
11   mypartner = xor(me, bit)
12   bit = shiftl(bit,1)
13   CALL sync_all()                     ! barrier
14   local_temp(:) = x(:)[mypartner]
15   CALL sync_all()                     ! barrier
16   x(:) = x(:) + local_temp(:)
17 ENDDO
```

butterfly reduction pattern

remote memory access

Listing 1: Sum reduction of arrays in CAF.

at end, all values of x are identical

CAF was added to the Fortran standard in 2008

Unified Parallel C (UPC) [1999]

Extension to C

blocking factor

shared/global arrays (PGAS)

```
1  shared [N*N/THREADS] uint8_t orig[N][N], edge[N][N];
2  int Sobel() {
3    int i,j,d1,d2;
4    double magnitude;
5    //                 init   cond    step  affinity
6    upc_forall(i=1;   i<N-1; i++;  &edge[i][0]) {
7      for(j=1; j<N-1; j++) {
8        d1  =  (int) orig[i-1][j+1] - orig[i-1][j-1];
9        d1 += ((int) orig[i  ][j+1] - orig[i  ][j-1]) << 1;
10       d1 +=  (int) orig[i+1][j+1] - orig[i+1][j-1];
11       d2  =  (int) orig[i-1][j-1] - orig[i+1][j-1];
12       d2 += ((int) orig[i-1][j  ] - orig[i+1][j  ]) << 1;
13       d2 +=  (int) orig[i-1][j+1] - orig[i+1][j+1];
14       magnitude = sqrt(d1*d1+d2*d2);
15       edge[i][j] = magnitude>255 ? 255 : (uint8_t)magnitude;
16     }
17   }
18   if (MYTHREAD == 0)
19     printf("DONE\n");
20
21   return 0;
22 }
```

Listing 3: Parallel edge detection using Sobel operators in UPC.

parallel for-loop

threads only execute iterations where affinity is local

SPMD and remote data accesses

explicit thread ID check

UPC is still used, with multiple distributions

DARPA HPCS Program

- High Productivity Computing Systems (HPCS)
- Launched in 2002 with five teams (later narrowed to three)
  - Cray, HP, IBM, SGI, Sun
- Language efforts
  - X10 [IBM, 2004]
    - Based on Java runtime environment
  - Fortress [Sun, 2008]
    - Unique idea: "typesetting" code
    - Discontinued in 2012 due to type system complications
  - Chapel [Cray, 2009]
    - "Cascade High Productivity Language"

X10

Asynchronous PGAS

```
1  val initializer = (i:Point) => {
2    val r = new Random();
3    var local_result:double = 0.0D;
4    for (c in 1..N) {
5      val x = r.nextDouble();
6      val y = r.nextDouble();
7      if ((x*x + y*y) <= 1.0)
8        local_result++;
9    }
10   local_result
11 };
12 val result_array = DistArray.make[Double](Dist.makeUnique(), initializer);
13 val sum_reducer = (x:Double, y:Double) => { x + y };
14 val pi = 4 * result_array.reduce(sum_reducer, 0.0) / (N * Place.MAX_PLACES);
```

Listing 6: Estimating π using Monte Carlo method in X10.

X10 is still used, but seems to have lost momentum

Fortress

Hybrid async PGAS and implicit parallelism

```
spawn x.region do
    f(x)
end
```

Computes f(x) wherever x is currently stored

$$\Sigma \quad \Pi$$

Valid operators

```
1  var a : RR64 = 0.0
2  var b : RR64 = 0.0
3  var c : RR64 = 0.0
4
5  DELTA = b^2 - 4 a c
6  x_1 = (-b - SQRT DELTA)/(2 a)
7  x_2 = (-b + SQRT DELTA)/(2 a)
```

(a) Small example program in Fortress without unicode characters.

$$\begin{aligned}
&\text{var } a : \mathbb{R}64 = 0.0\\
&\text{var } b : \mathbb{R}64 = 0.0\\
&\text{var } c : \mathbb{R}64 = 0.0\\
&\Delta = b^2 - 4\,a\,c\\
&\mathrm{x}_1 = \frac{-b - \sqrt{\Delta}}{2\,a}\\
&\mathrm{x}_2 = \frac{-b + \sqrt{\Delta}}{2\,a}
\end{aligned}$$

(b) Small example program in Fortress that supports unicode characters.

Officially discontinued in 2012; source code is still available

Chapel

- New language designed for parallel computation
  - Heavily influenced by ZPL and High-Performance Fortran
- Design is based on user requirements
  - Recent graduates: "a language similar to Python, Matlab, Java, etc."
  - HPC veterans: "a language that gives me complete control"
  - Scientists: "a language that lets me focus on the science"

Chapel

- Chapel stated goals:
  - *"A language that lets scientists **express** what they want …*
  - *… without taking away the **control** that veterans want …*
  - *… in a package that's as **attractive** as recent graduates want."*

Chapel themes

- Open source compiler (Apache license)
  - Uses Pthreads for local concurrency
  - Uses GASNet library for distributed communication
- Multi-resolution parallelism
  - Multiple levels of abstraction (task and data parallelism)
  - Higher levels build on lower levels
  - Developers can mix-and-match as desired
- Locality control
  - PGAS memory model; developers control data locales
- Reduced gap between HPC and mainstream
  - Type inference, generic programming, optional OOP

Chapel examples

```
var done: bool = true;     // 'done' is a boolean variable, initialized to 'true'

proc abs(x: int): int {    // a procedure to compute the absolute value of 'x'
  if (x < 0) then
    return -x;
  else
    return x;
}

var Hist: [-3..3] int,               // a 1D array of integers
    Mat: [0..#n, 0..#n] complex,     // a 2D array of complexes
    Tri: [i in 1..n] [1..i] real;    // a "triangular" skyline array

var count = 0;             // '0' is an integer, so 'count' is too
const area = 2*r;          // if 'r' is an int/real/complex, 'area' will be too
var len = computeLen();    // 'len' is whatever type computeLen() returns
config const n = 10;       // can be overridden by "--n=X" on the command line

for i in 1..n do           // print 1, 2, 3, ..., n
  writeln(i);

for elem in Mat do         // increment all elements in Mat
  elem += 1;
```

Chapel examples

domain definition

```
1  const BigD = {0..n+1, 0..n+1} dmapped Block(boundingBox=[0..n+1, 0..n+1]),
2            D: subdomain(BigD) = {1..n, 1..n};
3  var A, Temp: [BigD] real;
4
5  do {
6    forall (i,j) in D do
7      Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;
8    const delta = max reduce abs(A[D] - Temp[D]);
9    A[D] = Temp[D];
10 } while (delta > epsilon);
```

implicit data parallelism

average neighbors' values

Listing 4: Jacobi iteration example in Chapel (data parallel).

arbitrary domain array parameter

```
1  proc quickSort(arr: [?D],
2                 thresh = log2(here.numCores()), depth = 0,
3                 low: int = D.low, high: int = D.high) {
4    if high - low < 8 {
5      bubbleSort(arr, low, high);
6    } else {
7      const pivotVal = findPivot(arr, low, high);
8      const pivotLoc = partition(arr, low, high, pivotVal);
9      serial(depth >= thresh) do cobegin {
10       quickSort(arr, thresh, depth+1, low, pivotLoc-1);
11       quickSort(arr, thresh, depth+1, pivotLoc+1, high);
12 } } }
```

explicit task parallelism

Listing 5: Parallel Quicksort example in Chapel (task parallel).

Comparing languages

Partitioned Global Address Space Languages

MATTIAS DE WAEL, STEFAN MARR, BRUNO DE FRAINE, TOM VAN CUTSEM, and WOLFGANG DE MEUTER, Vrije Universiteit Brussel, Belgium

The Partitioned Global Address Space (PGAS) model is a parallel programming model that aims to improve programmer productivity while at the same time aiming for high performance. The main premise of PGAS is that a globally shared address space improves productivity, but that a distinction between local and remote data accesses is required to allow performance optimizations and to support scalability on large-scale parallel architectures. To this end, PGAS preserves the global address space while embracing awareness of non-uniform communication costs.

Today, about a dozen languages exist that adhere to the PGAS model. This survey proposes a definition and a taxonomy along four axes: how parallelism is introduced, how the address space is partitioned, how data is distributed among the partitions and finally how data is accessed across partitions. Our taxonomy reveals that today's PGAS languages focus on distributing regular data and distinguish only between local and remote data access cost, whereas the distribution of irregular data and the adoption of richer data access cost models remain open challenges.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Concurrent, distributed, and parallel languages; D.3.3 [**Language Constructs and Features**]: Concurrent programming structures

General Terms: Design, Languages

Additional Key Words and Phrases: Parallel programming, HPC, PGAS, message passing, one-sided communication, data distribution, data access, survey

ACM Reference Format:
Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, Wolfgang De Meuter. Partitioned Global Address Space Languages. *ACM Comput. Surv.* x, x, Article x (January 2015), 29 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

Execution models

- Fully SPMD
  - Fixed number of threads spawn at launch and diverge based on thread index checks (similar to MPI)
- Asynchronous PGAS
  - Single main thread; worker threads spawn automatically in marked parallel regions (similar to OpenMP)
- Fully Implicit
  - Threads spawned dynamically by runtime system as appropriate; no explicit parallel regions

Topologies and data access

- Topologies
  - Flat (indexed)
  - Rectangular / hypercube / torus / mesh
  - Hierarchical
- Access cost function
  - Two-level (local vs. remote)
  - Multi-level
- Data distribution
  - Implicit vs. explicit
  - Regular vs. irregular (domain uniformity)
- Remote data accesses
  - Implicit vs. explicit
  - Local vs. global

PGAS Language Summary

Language	Parallel Execution	Topology	Data Distribution	Distributed Data	Remote Access	Array Indexing
Retrospective PGAS languages						
HPF	Implicit	User defined mesh	Explicit	Regular	Implicit	Global
ZPL	Implicit	User defined mesh	Implicit	Regular	Explicit	Global
GA	SPMD	Flat ordered set	Explicit	Regular	Explicit	Global
Original PGAS languages						
CAF	SPMD	User defined mesh	Implicit	Regular	Explicit	Local
Titanium	SPMD	Flat ordered set	Explicit	Irregular	Expl. + Impl.	not applicable
UPC	SPMD	Flat ordered set	Explicit	Reg. + Irreg.	Implicit	Global
HPCS PGAS languages						
Chapel	APGAS + Impl.	User defined mesh	Explicit	Reg. + Irreg.	Expl. + Impl.	Global
X10	APGAS	Flat ordered set	Explicit	Reg. + Irreg.	Explicit	Global
Fortress	APGAS + Impl.	Hierarchical	Explicit	Reg. + Irreg.	Expl. + Impl.	Global

lower ≈ newer

Lessons learned??

For more details and full paper:
`http://dl.acm.org/citation.cfm?id=2716320`

Julia

- New dynamic language for numeric computing
  - Combines ideas from Python, Matlab, R, and Fortran
  - Mantra: "*vectorize when it feels right*"
  - Core is implemented in C/C++, JIT-compiled to native machine code
  - Includes a REPL
  - IJulia browser-based graphical notebook interface
- Goal: never make the developer resort to using two languages
  - Similar philosophy in Chapel community

```
nheads = @parallel (+) for i=1:100000000
  int(randbool())
end
```

Simulate coin tosses in parallel

```
function mandelbrot(z)
    c = z
    maxiter = 80
    for n = 1:maxiter
        if abs(z) > 2
            return n-1
        end
        z = z^2 + c
    end
    return maxiter
end
```

Calculate Mandelbrot function

Python for HPC

- Primary strength: writeability
  - Easy-to-learn
  - Low overhead and boilerplate
- Secondary strength: libraries & frameworks
  - NumPy (supports large, multi-dimensional matrices)
  - SciPy (scientific computing library that uses NumPy)
  - SageMath (open source Mathematica/Matlab alternative)
  - IPython (interactive parallel computing)
  - Many others!

Holy Grail impossible?

Challenge: design your own parallel language!

What would it look like?

For Thursday

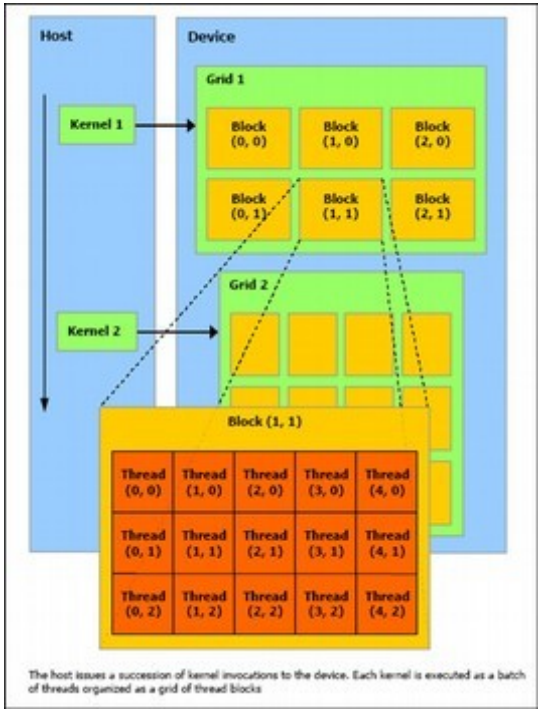
- Now: a CUDA crash course!

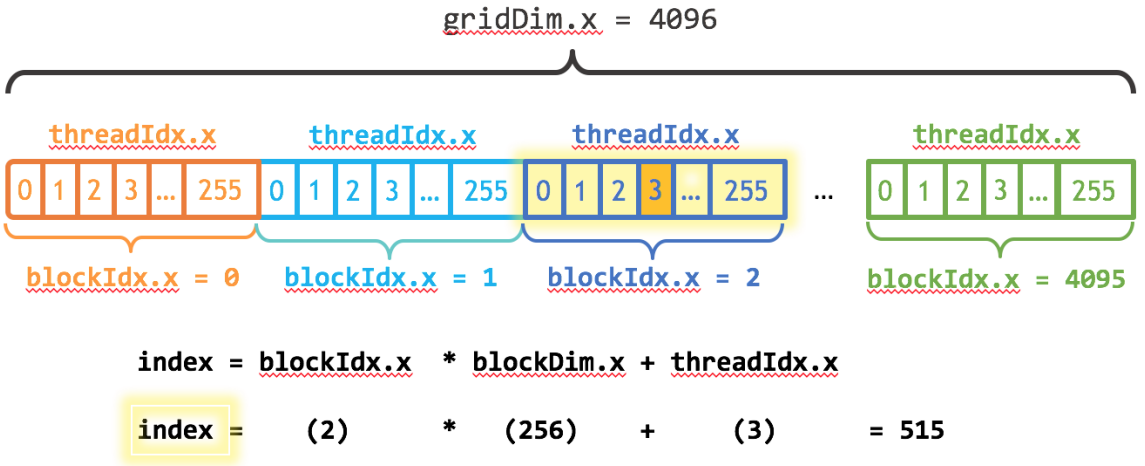
CUDA crash course

- CUDA: NVIDIA's GPU computation API for C++
  - Compile .cu source files with NVIDIA compiler (`nvcc`)
- Allocate memory on device manually
  - Traditional model: copy back and forth
  - Newer model: unified memory (like PGAS)
    - Call `cudaMallocManaged()` to allocate unified memory
- Many-way parallelism
  - Writes a kernel routine to be run on each thread (like Pthreads)
  - Must manually split up work among threads (arranged in a grid of blocks)
    - Common approach: grid-stride loop
- Device runs many threads in blocks
  - Call kernel: `kernel_func<<<numBlocks, blockSize>>>()`
  - Call `cudaDeviceSynchronize()` to wait for a kernel to finish

CUDA crash course

- Grid-stride access in kernel loops
  - Threads skip numBlocks * blockSize each iteration
  - Generalizes easily to any data size
  - Minimal locality impact because of faster GPU memory





From https://devblogs.nvidia.com/even-easier-introduction-cuda/

CUDA example

```
__global__
void add(int n, float *x, float *y)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride) {
    y[i] = x[i] + y[i];
  }
}

int main(void)
{
  int N = 1<<20;

  // unified memory – accessible from CPU or GPU
  float *x, *y;
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // run kernel on the GPU
  int blockSize = 256;
  int numBlocks = N / blockSize;
  add<<<numBlocks, blockSize>>>(N, x, y);

  // wait for GPU to finish
  cudaDeviceSynchronize();

  // check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++) {
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  }
  printf("Max error: %f\n", maxError);

  // free memory
  cudaFree(x);
  cudaFree(y);

  return 0;
}
```

From https://devblogs.nvidia.com/even-easier-introduction-cuda/