ME964
High Performance Computing for Engineering Applications

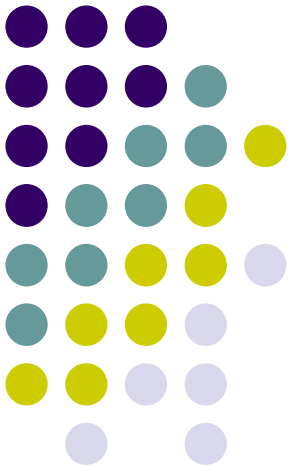
CUDA Optimization:

Execution Configuration Heuristics

Instruction Optimization

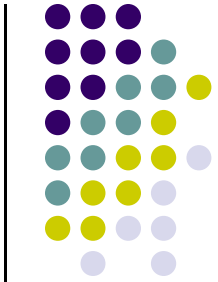
Parallel Prefix Scan Example

March 15, 2012

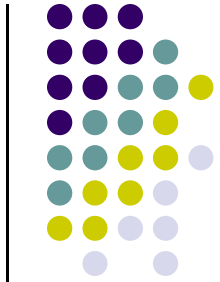


"Adding manpower to a late software project makes it later".
Fred Brooks

Before We Get Started…



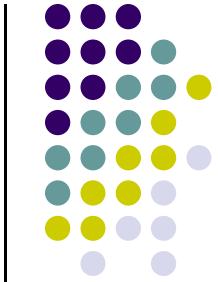
- Last time
  - Tiling – a programming design pattern in CUDA
  - CUDA Optimization/Best Practices issues
    - Example: Vector Reduction – a journey through seven layers of optimization
- Today
  - Following up on your feedback
  - CUDA Optimization Issues
    - Execution Configuration & Instruction Execution Issues
    - CUDA Optimization wrap up: high, medium, and low priority optimization recommendations
    - Parallel Prefix Scan Example
- Other issues
  - HW7 due tonight at 11:59 PM
  - HW8: posted shortly. The last of a sequence of harder assignments
  - Midterm Project, default choice: info emailed out and posted online
    - Intermediate progress report: Due on March 29
    - Strategy: get something to work, even if it's far from optimal. Improve on this preliminary design in case you decide to make this the topic of your Final Project



CUDA Optimization:

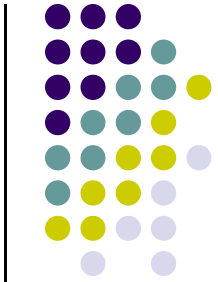
Execution Configuration Heuristics

Blocks per Grid Heuristics



- # of blocks > # of stream multiprocessors (SMs)
  - If this is violated, then you'll have idling SMs
- # of blocks / # SMs > 2
  - Multiple blocks can run concurrently on a multiprocessor
  - Blocks that aren't waiting at a `__syncthreads()` keep the hardware busy
  - Subject to resource availability – registers, shared memory
- # of blocks > 100 to scale to future devices
  - Blocks waiting to be executed in pipeline fashion
  - To be on the safe side, 1000's of blocks per grid will scale across multiple generations
  - If you bend backwards to meet this requirement maybe GPU not the right choice

Threads Per Block Heuristics

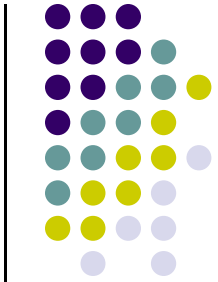


- Choose threads per block as a multiple of warp size
  - Avoid wasting computation on under-populated warps
  - Facilitates coalescing

- Heuristics
  - Minimum: 64 threads per block
    - Only if multiple concurrent blocks
  - 192 or 256 threads a better choice
    - Usually still enough registers to compile and invoke successfully
  - This all depends on your computation, so experiment!

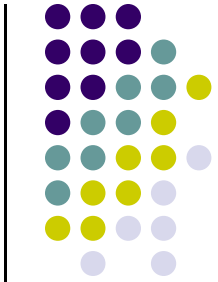
- Always use the `nvvp` profiler to understand how many registers you used, what bandwidth you reached, etc.

Occupancy



- In CUDA, executing other warps is the only way to hide latencies and keep the hardware busy
- Occupancy = Number of warps running concurrently on a SM divided by maximum number of warps that can run concurrently
  - When adding up the number of warps, they can belong to different blocks
- Can have up to 48 warps managed by one Fermi SM
  - For 100% occupancy your application should run with 48 warps on an SM
- Many times one can't get 48 warps going due to hardware constraints

CUDA Optimization: A Balancing Act

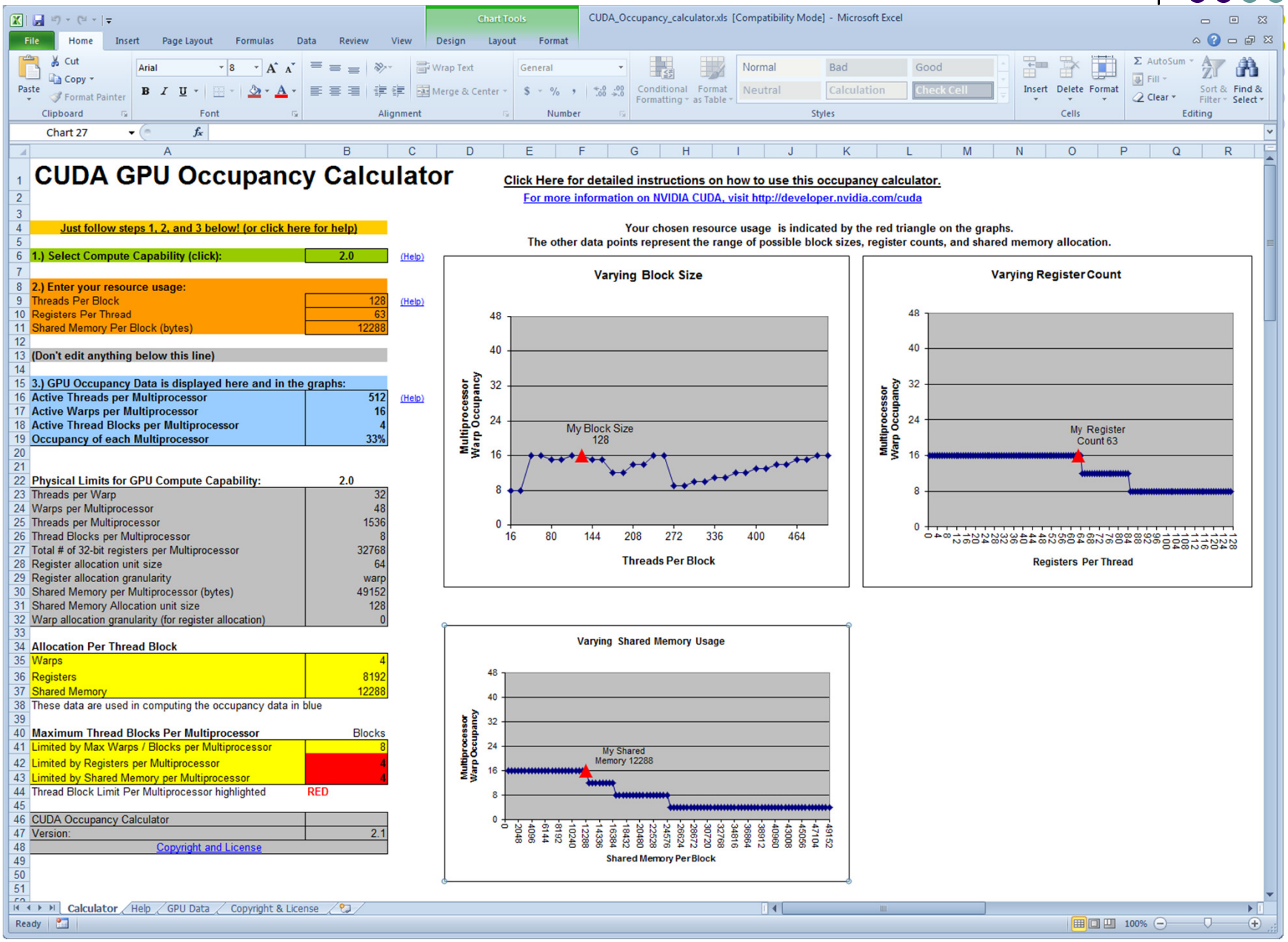


- Hardware constraints:
  - Number of registers per kernel
    - 32K per multiprocessor, partitioned among concurrent threads active on the SM
  - Amount of shared memory
    - 16 or 48 KB per multiprocessor, partitioned among SM concurrent blocks

- Use `-maxrregcount=N` flag on `nvcc`
  - `N` = desired maximum registers / kernel
  - At some point “spilling” into local memory may occur
    - Might not be that bad, there is L1 cache that helps to some extent

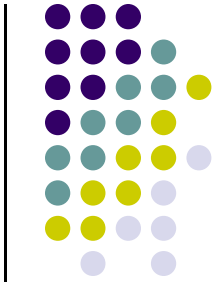
- Recall that you cannot have more than 8 blocks executed by one SM

NVIDIA CUDA Occupancy Calculator



Google: “cuda occupancy calculator”

Occupancy != Performance

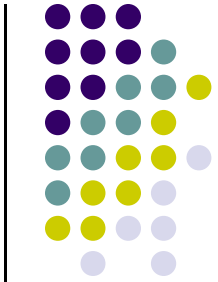


- Increasing occupancy does not necessarily increase performance
  - If you want to read more about this, there is a Volkov paper on class website
  - What comes to the rescue is the Instruction Level Parallelism (ILP) that becomes an option upon low occupancy

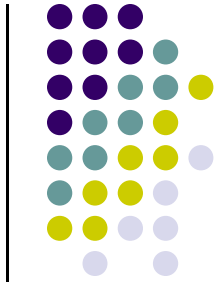
YET, OFTEN TIMES IS A BAD OMEN …

- Low-occupancy multiprocessors are likely to have a hard time when it comes to hiding latency on memory-bound kernels
  - This latency hiding draws on Thread Level Parallelism (TLP); i.e., having enough threads (warps, that is) that are ready for execution

Parameterize Your Application

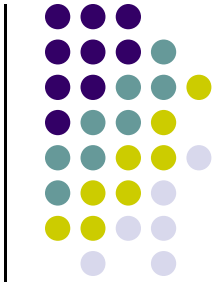


- Parameterization helps adaptation to different GPUs
- GPUs vary in many ways
  - # of SMs
  - Memory bandwidth
  - Shared memory size
  - Register file size
  - Max. threads per block
  - Max. number of warps per SM
- You can even make apps self-tuning (like FFTW and ATLAS)
  - “Experiment” mode discovers and saves optimal configuration



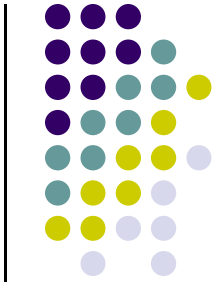
CUDA Instruction Optimization

Need for This Discussion…



- We discussed at length about what you can do to operate at a memory effective bandwidth close to the nominal bandwidth
- Discuss for 10 minutes about instruction execution throughput
- In the rare situation you have high arithmetic intensity, it's good to know how fast math gets executed on the device

Instruction Throughput
[How to Evaluate It]



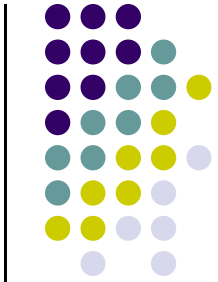
- Throughputs typically given in number of operations per clock cycle per SM
  - Called “nominal throughput” for reasons explained shortly
- Note that for a warp size of 32, one instruction results in 32 operations
  - You have 32 threads operating in lockstep fashion
- Assume that for a given SM, T is the nominal throughput of operations per clock cycle for a certain math instruction
  - Then, 32 operations; i.e., that “certain math instruction” will take x clock cycles, where

$$x = 32/T \text{ [instructions/clock-cycle]}$$

- Concluding: instruction throughput – one instruction every 32/T clock cycles
  - In theory, even better: this number gets multiplied by # of SMs in your GPU
- Quick Remark: the higher the T (operations/clock-cycle), the better

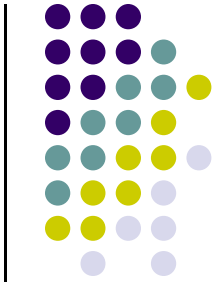
Nominal Throughputs for Native Arithmetic Instructions

[Operations per Clock Cycle per Multiprocessor]



Throughput of Native Arithmetic Instructions	Compute Capability 1.x	Compute Capability 2.0	Compute Capability 2.1
32-bit floating-point add, multiply, multiply-add	8	32	48
64-bit floating-point add, multiply, multiply-add	1	16	4
32-bit integer add, logical operation	8	32	48
32-bit integer shift, compare	8	16	16
32-bit integer multiply, multiply-add, sum of absolute difference	Multiple instructions	16	16
24-bit integer multiply (__[u]mul24)	8	Multiple instructions	Multiple instructions
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (__log2f), base-2 exponential (exp2f), sine (__sinf), cosine (__cosf)	2	4	8
Type conversions	8	16	16

CUDA Instruction Performance



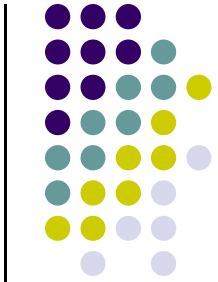
- Instruction performance (per warp), depends on
  - Operand read cycles
  - Nominal instruction throughput
  - Result update cycles

This is what we just discussed on the previous two slides

- In other words, instruction performance depends on
  - Nominal instruction throughput
  - Memory latency
  - Memory bandwidth

- "Cycle" refers to the multiprocessor clock rate
  - 1.4 GHz on the GTX480, for example

Runtime Math Library: A Word on Fast Math…

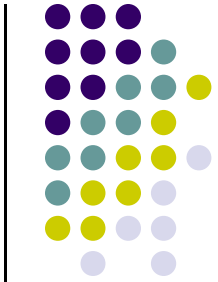


- There are two types of runtime math operations
  - `__funcf()`: direct mapping to hardware ISA
    - Fast but lower accuracy (see programming guide for details)
    - Examples: `__sinf(x)`, `__expf(x)`, `__powf(x,y)`
  - `funcf()` : compile to multiple instructions
    - Slower but higher accuracy
    - Examples: `sinf(x)`, `expf(x)`, `powf(x,y)`

- The **`-use_fast_math`** compiler option forces every `funcf()` to compile to `__funcf()`

GPU Results May Not Match CPU

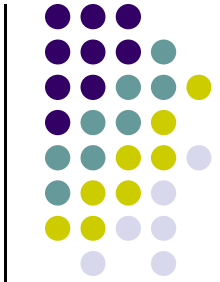
[Two Slide Detour: 1/2]



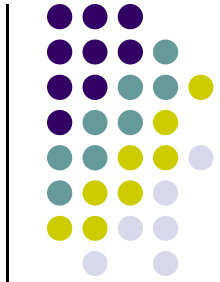
- Many variables: hardware, compiler, optimization settings
  - CPU operations aren't strictly limited to 0.5 ulp
    - Sequences of operations can be more accurate due to 80-bit extended precision ALUs
    - ULP: "Unit in the Last Place" is the spacing between floating-point numbers, i.e., the value that the least significant bit (lsb) represents if it is 1.
      - It is used as a measure of precision in numeric calculations
- Floating-point arithmetic is not associative

FP Math is Not Associative!

[Two Slide Detour: 2/2]

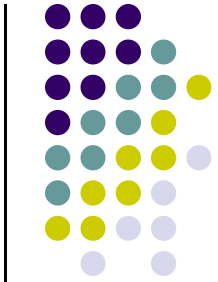


- In symbolic math, `(x+y)+z == x+(y+z)`
- This is not necessarily true for floating-point addition
- When you parallelize computations, you likely change the order of operations
  - Round off error propagates differently
- Parallel results may not exactly match sequential results
  - This is not specific to GPU or CUDA – inherent part of parallel execution



CUDA Optimization: Wrap Up…

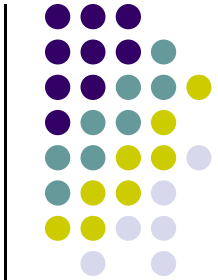
Performance Optimization
[Wrapping Up…]



- We discussed many rules and ways to write better CUDA code
- The next several slides sort this collection of recommendations based on their importance
- Writing CUDA software is a craft/skill that is learned
  - Just like playing a game well: know the rules and practice
  - A list of high, medium, and low priority recommendations wraps up discussion on CUDA optimization
    - For more details, check the CUDA C Best Practices Guide:

http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf

Writing CUDA Software: High-Priority Recommendations



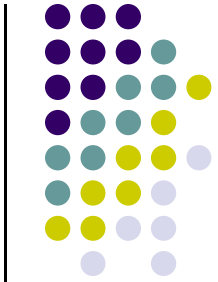
1. To get the maximum benefit from CUDA, focus first on finding ways to parallelize sequential code

2. Use the effective bandwidth of your computation as a metric when measuring performance and optimization benefits

3. Minimize data transfer between the host and the device, even if it means running some kernels on the device that do not show performance gains when compared with running them on the host CPU

http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf

Writing CUDA Software: High-Priority Recommendations



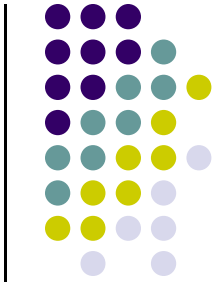
4. Ensure global memory accesses are coalesced whenever possible

5. Minimize the use of global memory. Prefer shared memory access where possible (consider tiling as a design solution)

6. Avoid different execution paths within the same warp

http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf

Writing CUDA Software: Medium-Priority Recommendations



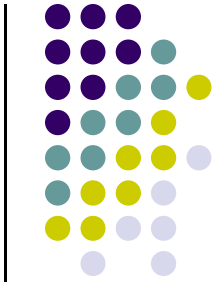
1. Accesses to shared memory should be designed to avoid serializing requests due to bank conflicts

2. To hide latency arising from register dependencies, maintain sufficient numbers of active threads per multiprocessor (i.e., sufficient occupancy)

3. The number of threads per block should be a multiple of 32 threads, because this provides optimal computing efficiency and facilitates coalescing

http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf

Writing CUDA Software: Medium-Priority Recommendations



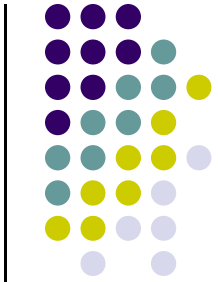
4. Use the fast math library whenever speed and you can live with a tiny loss of accuracy

5. Prefer faster, more specialized math functions over slower, more general ones when possible

6. Use signed integers rather than unsigned integers as loop counters
   - The compiler can optimize more aggressively with signed arithmetic than it can with unsigned arithmetic (due to rules regarding overflow behaviour). This is of particular note with loop counters

http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf

Writing CUDA Software: Low-Priority Recommendations



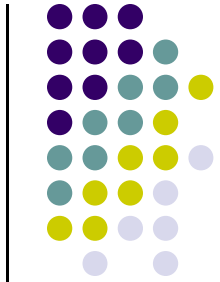
1. For kernels with long argument lists, place some arguments into constant memory to save shared memory

2. Use shift operations to avoid expensive division and modulo calculations

3. Avoid automatic conversion of doubles to floats

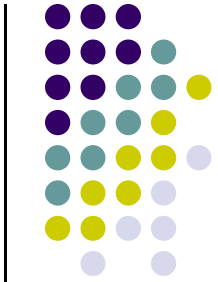
4. Make it easy for the compiler to use branch predication in lieu of loops or control statements

http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf



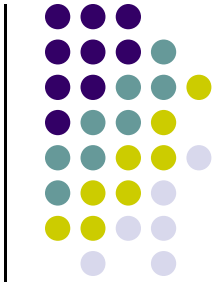
Example:
Parallel Prefix Scan on the GPU

Software Design Exercise: Parallel Prefix Scan



- Vehicle for software design exercise: parallel implementation of prefix sum
  - Serial implementation – assigned as HW
  - Parallel implementation: topic of assignment #8
- Goal 1: Flexing our CUDA muscles
- Goal 2: Understand that
  - Different algorithmic designs lead to different performance levels
  - Different constraints dominate in different applications and/or design solutions
- Goal 3: Identify design patterns that can result in superior parallel performance
  - Understand that there are patterns and it's worth being aware of them
  - To a large extend, patterns are shaped up by the underlying hardware

Parallel Prefix Sum (Scan)



- Definition:

  The all-prefix-sums operation takes a binary associative operator ⊕ with identity ***I***, and an array of n elements

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

  and returns the ordered set

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

- Example:
  If ⊕ is addition, then scan on the set

  [3 1 7 0 4 1 6 3]

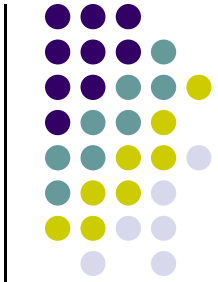
  returns the set

  [0 3 4 11 11 15 16 22]

Exclusive scan: last input element is not included in the result

(From Blelloch, 1990, "Prefix Sums and Their Applications)

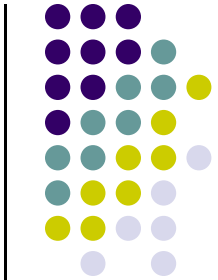
Scan on the CPU



```
void scan( float* scanned, float* input, int length)
{
  scanned[0] = 0;
  for(int i = 1; i < length; ++i)
  {
    scanned[i] = scanned[i-1] + input[i-1];
  }
}
```

- Just add each element to the sum of the elements before it
- Trivial, but sequential
  - Tempted to say that algorithms don't come more sequential than this…
- Requires exactly *n-1* adds

Applications of Scan



- Scan is a simple and useful parallel building block
  - Convert recurrences from sequential …

```
out[0] = f(0)
for(j=1;j<n;j++)
    out[j] = out[j-1] + f(j);
```

  - … into parallel:

```
forall(j) in parallel
    temp[j] = f(j);
scan(out, temp);
```

- Useful in implementation of several parallel algorithms:

• Radix sort	• Polynomial evaluation
• Quicksort	• Solving recurrences
• String comparison	• Tree operations
• Lexical analysis	• Histograms
• Stream compaction	• Etc.