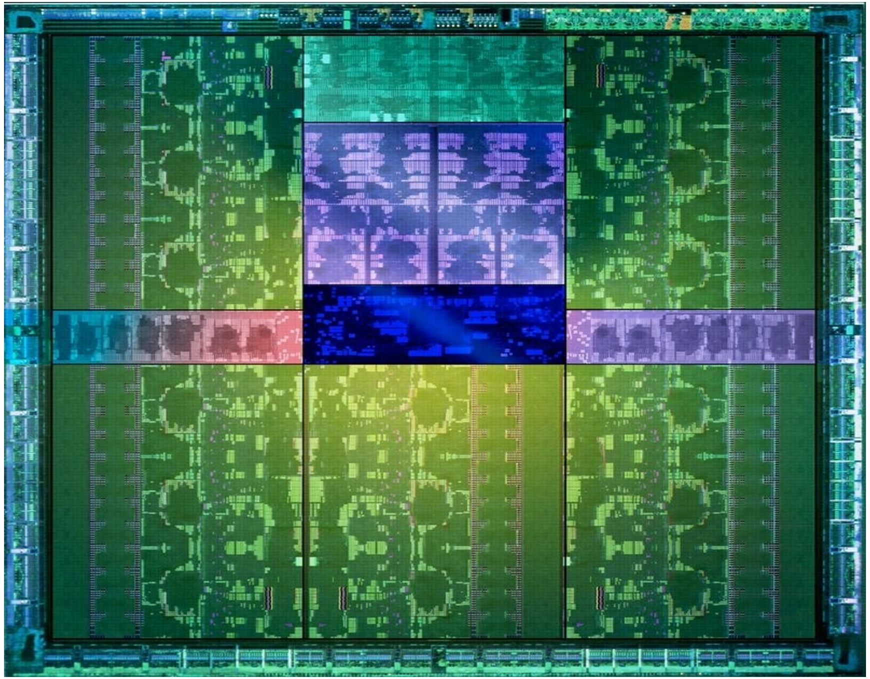
Contents

- Comparison of CPUs and GPUs
- Programming Styles
- What is Easy to Accelerate?
- Libraries

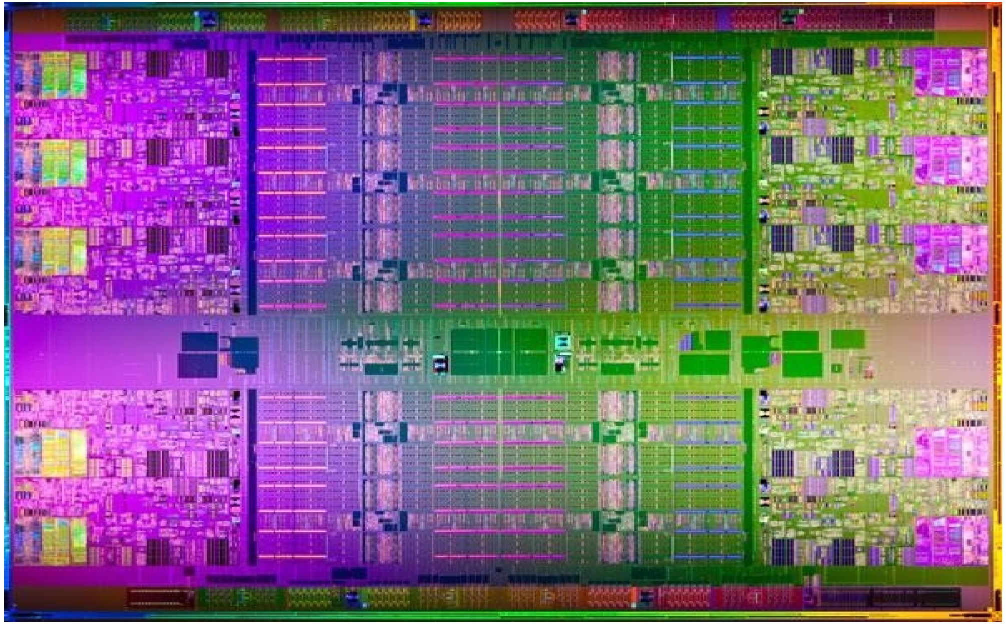
CPU or GPU?

Which die is the CPU, which one the GPU?

GK110



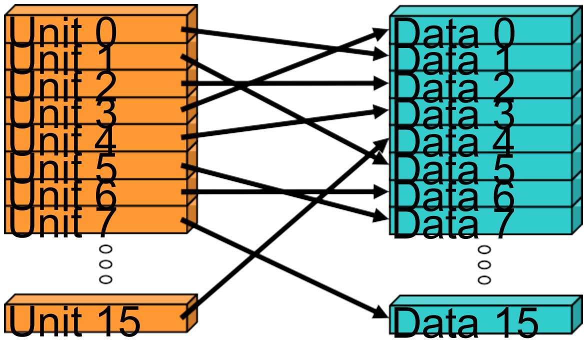
XEON-E7



Four Levels of Parallelism

SIMD 10x

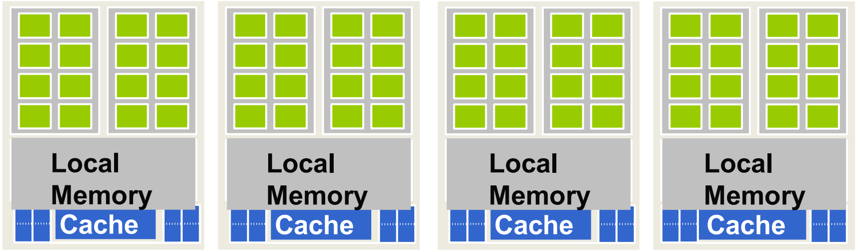
30x



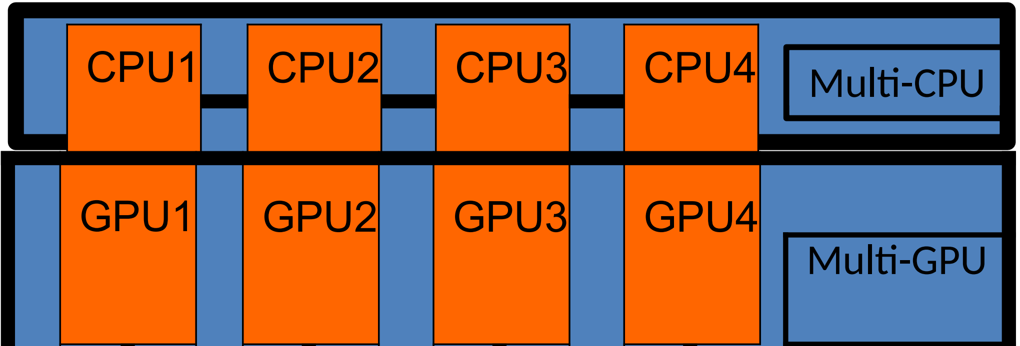
Many-core 10x

30x

↑·10↑

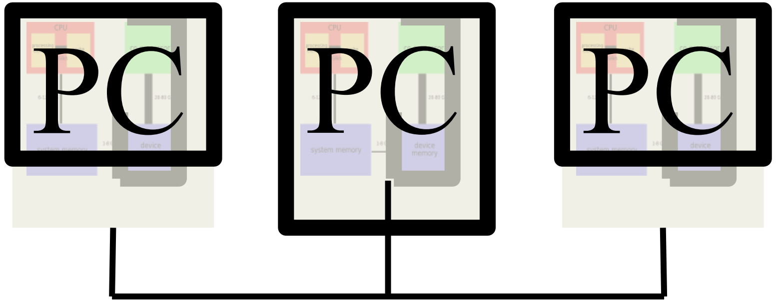


Intra-node 10x

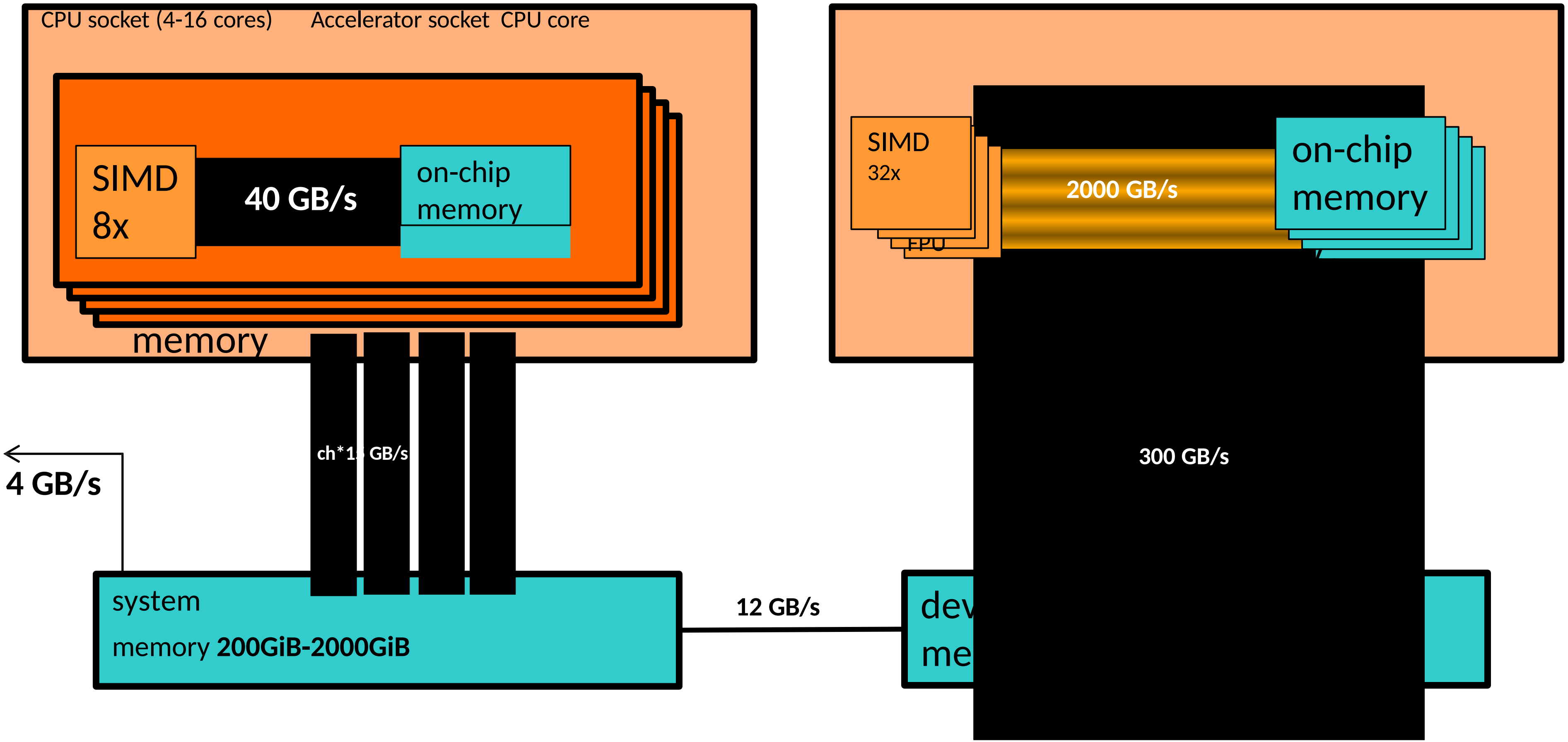


Inter-node 10^3x

↑·10↑



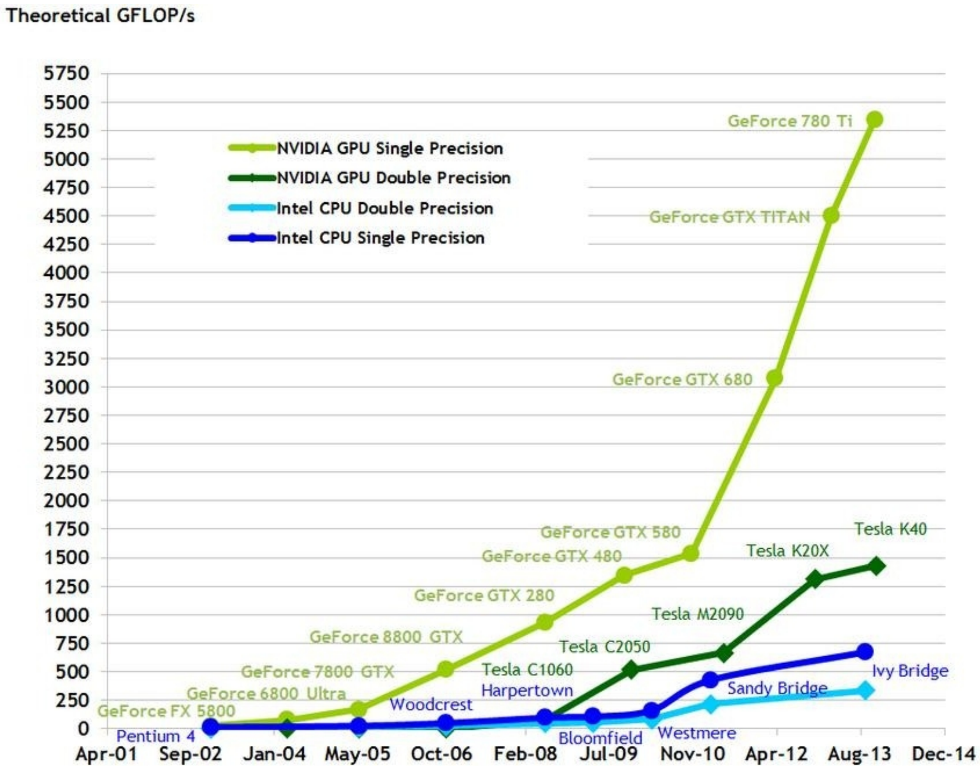
Bandwidth in an Accelerator System

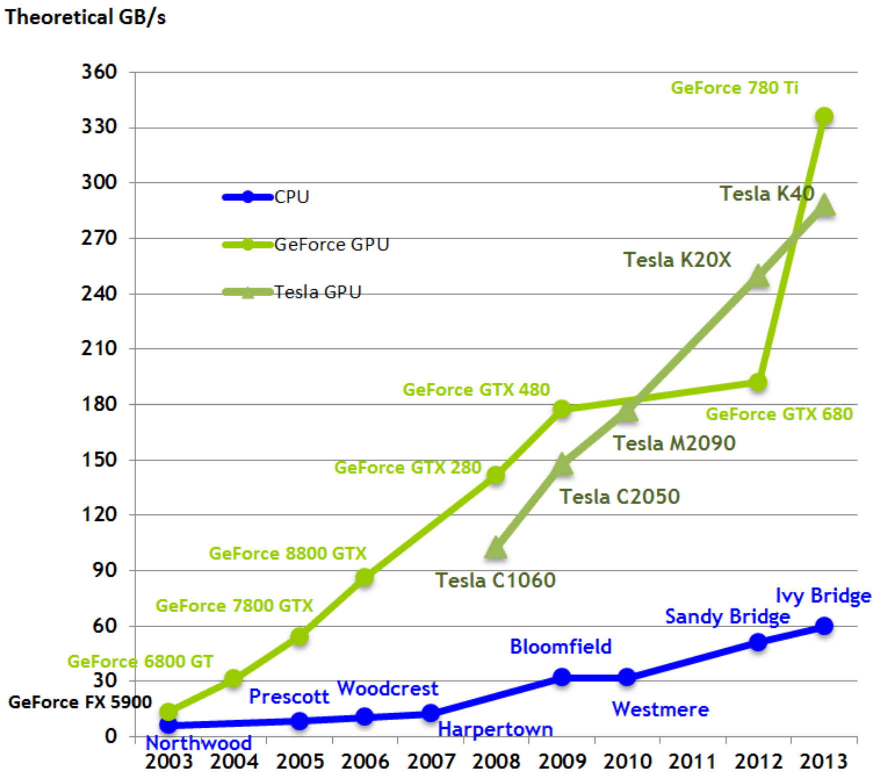


GPUs vs. CPUs

	Tesla K20	Xeon E7-4800 4P
Core count	13 SMs 64/832 (DP), 192/2,496 (SP)	10 Cores 2 FP-ALUs/core, SSE 16B
Frequency	0.7GHz	2.4GHz
Peak Compute Performance	1,165 GFLOPS (DP) 3,494 GFLOPS (SP)	96 GFLOPS (DP)
Use model	throughput-oriented	latency-oriented
Latency treatment	toleration	minimization
Programming	1000s-10,000s of threads	10s of threads
Memory bandwidth	250 GBytes/sec	34 GByte/s (per P)
Memory capacity	5 GB	up to 2TB
Die size	$550mm^2$	$684\ mm^2$
Transistor count	7.1 billion	2.3 billion
Technology	28nm	32nm
Power consumption	250W	130W
Power efficiency	4.66 GFLOPs/Watt (DP) 14 GFLOPs/Watt (SP)	0.74 GFLOPs/Watt (DP)

Theoretical Performance





Parallelism on CPUs and GPUs

CPU

- SIMD AVX 32B, Phi 64B
  - MADD with 8-16 floats MADD
  - with 4-8 doubles Coding: explicit,
  - automatic
- Minimum multi-threading
  - #threads=#cores (good)
  - #threads= 2*#cores (good)
  - #threads= 10*#cores (difficult)
  - Coding: explicit with resident threads, implicit with libraries

GPU

- SIMD warp size 32
  - MADD with 32 floats
  - MADD with 32 doubles
  - Coding: implicit, partly explicit
- Maximum multi-threading
  - #warps=#'cores'≈15 (bad)
  - #warps ≈100 (difficult)
  - #warps>1000 (good)
  - Coding: implicit with max. parallelism, explicit (advanced)

Memory on CPUs and GPUs

CPU

- Deep and large memories
  - Core: Reg, L1, L2
  - Shared: L3, eDRAM
  - Coding: implicit
- Usage
  - Optimize 1D locality
  - Optimize size of working sets
  - Prefetch, pipeline
  - Coding: implicit, explicit

GPU

- Smaller and specialized memories
  - Core: Reg, L1 or shmem
  - Shared: L2, constant
  - Coding: explicit, implicit with libs
- Usage
  - Optimize 1D, 2D, 3D locality
  - Decide on data location: Reg, L1, shmem, constant
  - Many warps and low latency vs. amount of local data per warp
  - Coding: explicit

Similarities and Differences

CPU

- For high performance
  - SIMD
  - Multi-threading
  - Memory access alignment
  - Minimal latency with large caches
  - Working set opt. wrt deep caches
  - Locality opt. wrt cache lines, NUMA
- Opt. serial performance
  - High normal and boost frequency
  - Low latency caches
  - Speculative execution

GPU

- For high performance
  - SIMD
  - Multi-threading
  - Memory access alignment
  - Minimal latency with many warps
  - Working set opt. wrt #warps
  - Locality opt. wrt memory types
- Opt. throughput performance
  - Lower normal and boost frequency
  - L2 latency is high
  - No speculative execution

CUDA Ecosystem

What is Easy to Accelerate?

Embarrassingly Parallel Loop

- ```
  for(int i=0; i<SIZE; ++i)
  {  c[i]= a[i+1]+b[i]*a[i-yoff];
  func(a,b,c,i);
  }
  ```
- Relevant for performance
  - `SIZE > 10k`
  - Arithmetic intensity
  - Regularity of memory access
  - Amount of local state and reuse
  ```

Branches in Loops

- ```
  for(int i=0; i<SIZE; ++i) {
    if(cond(i))special_func(a,b,c,i);
     else normal_func(a,b,c,i);
  }
  ```
- Only a problem if all these conditions hold:
  - Special case is more than 10% of cases
  - Normal and special case differ largely in execution times
  - Data of special cases is scattered in memory
  ```

Index Dependencies

-
```
for(int i=1; i<SIZE; ++i)
{   a[i]+= a[i-1];
}
```
- Replace serial dependence
  - Use equivalent parallel variant
  - If allowed, use approximate parallel variant
  - Check if parent computation can use other ingredients

Data Movement is Critical

- ```
  CPU_func1(a,b,c);
  GPU_func1(a,b,c); // implicit transfer!
  CPU_func2(a,b,c);
  GPU_func2(a,b,c); // implicit transfer!
  ```

- Such CPU-GPU alternation only works well if
  - Execution time of `GPU_func*` is at least a milisecond
  - High arithmetic intensity, e.g. matrix*matrix, not matrix*vector
- Otherwise
  - GPU must perform multiple operations on the same data, e.g. multiple vector-vector or matrix-vector operations.
  ```

Linked List

- `for(; elm!=nullptr; elm= elm->next) {`
  ` func(elm->data);`
  `}`
- Do not do this!
  - Unless all parallelism can be used efficiently in `func()`
  - Terrible performance on CPUs and GPUs
  - Vector almost always dramatically faster than list
  - Even `insert(pos)`,`delete(pos)` much faster in vector if we first search for `pos`

Data Structures on GPUs

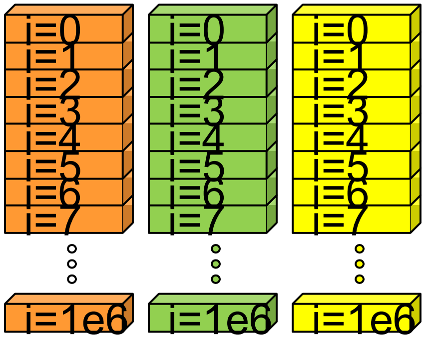
	Difficulty	Speed	Support	Format
vector	easy	fast	everywhere	contiguous data
dense matrix	easy	fast	many libs	contiguous data
sparse matrix	moderate	fast	many libs	**CSR**, BCSR, (B)CSC, COO, special
graph	moderate	fast	multiple libs	**CSR**, special
tree	difficult	fast	little	various special formats
list	moderate	slow	none	special formats

Where to Put the Parallelism?

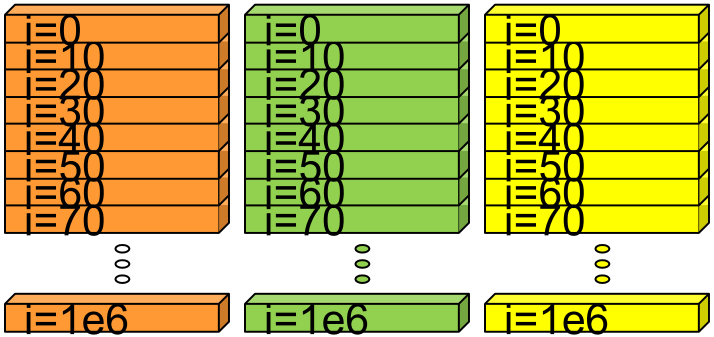
```
for(int i=0; i<SIZE; i+=block_size)
{   func1(a,b,c,i,block_size);      }
for(int i=0; i<SIZE; i+=block_size)
{   func2(a,b,c,i,block_size);      }
for(int i=0; i<SIZE; i+=block_size) {

    func3(a,b,c,i,block_size);
}
```

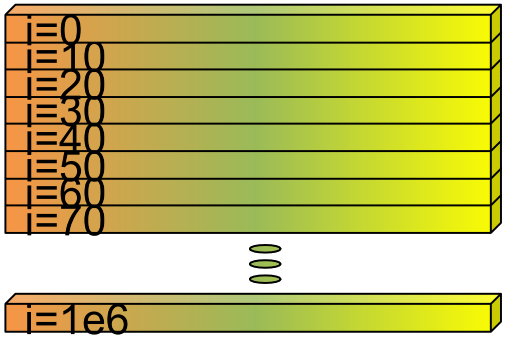
3 loops, block_size=1



3 loops, block_size=10



1 loop, block_size=10



Libraries

Summary

- For high performance on CPUs and GPUs
  - High parallelism and high data locality
  - Optimizations are similar in concept, but different and very involved in detail
  - Difficult to do by hand → use libraries
- What is easy to accelerate?
  - Large loops with no/simple index dependencies
  - Data placement and movement are crucial