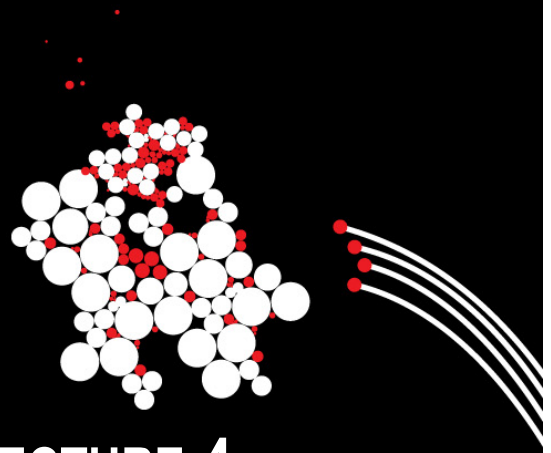
UNIVERSITY OF TWENTE.

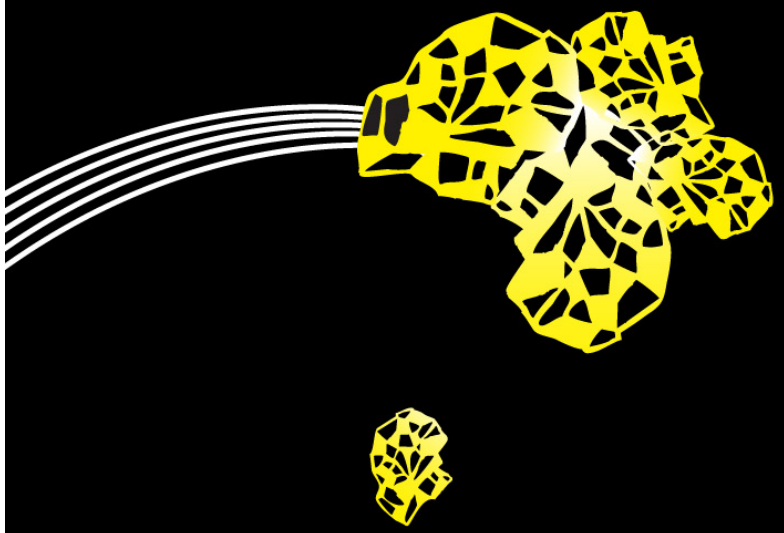


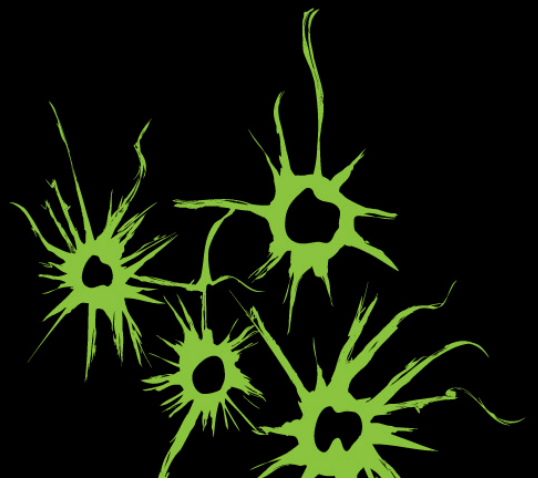
CONCURRENT PROGRAMMING – LECTURE 4

VECTOR PROGRAMMING: OPENMP AND OPENCL

MODULE 8: PROGRAMMING PARADIGMS

23 MAY 2019

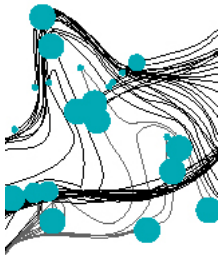




CONCURRENT PROGRAMMING

CONTENTS

2	Tue 30/4	Basics of concurrency, thread safety, testing concurrent systems	Ch. 1-3, 12 (+ SS material)
3	Tue 7/5	Synchronisation	Ch. 4, 5, 13, 14.1-14.4
4	Wed 15/5	Liveness, performance, and fairness	Ch. 10,11
5	Thu 23/5	Homogeneous threading (OpenMP, OpenCL)	Papers
6	Mon 3/6	Safe concurrency (Software Transactional Memory, Rust)	Papers
7	Fri 7/6	Fine-grained concurrency, memory models	Ch. 14.5-6, 15, 16



CONCURRENT PROGRAMMING

LECTURE 4

- Quiz on liveness, fairness and performance
- Heterogeneous vs. homogeneous threading
- Program dependences
- OpenMP: Parallellisation by compiler directives
- OpenCL: General-purpose GPU programming

Literature:

- Selected sections 4.3 – 4.8 from ‘Using OpenMP’
- Section 5-CP.1 and 5-CP.3 from course manual
- DrDobbs tutorial on OpenCL

QUESTION 1

```
private Object lockSec = new Object();
private Object lockMin = new Object();
private int seconds; //@ guardedby lockSec
private int minutes; //@ guardedby lockMin
```

```
public void addSeconds(int s) {
   synchronized (lockSec) {
      synchronized (lockMin) {
         seconds = (seconds + s) % 60;
         int t = (seconds + s)/60;
         minutes = minutes + t;
      }
   }
}
```

```
public void reset() {
   synchronized (lockMin) {
      synchronized (lockSec) {
         minutes = 0;
         seconds = 0;
      }
   }
}
```

This program has a problem?
What is it, and how to solve it?

a. Deadlock, remove all locks
b. Deadlock, change lock order
c. Livelock, change lock order
d. Starvation, remove reset method

QUESTION 2

How to improve the performance while preserving thread safety?

a. Remove the locks
b. Use explicit locks
c. Use multiple locks (lock splitting)
d. Lock coarsening

```
public class Clock {
    private int sec;
    private int min;
    private int colour;

    public synchronized void updateTime(int s, int m) {
        int newSec = sec + s;
        int newMin = min + m;
        sec = newSec % 60;
        min = (newSec/60 + newMin) % 60;
    }
    public synchronized void changeBackground(int c) {
        colour = c;
    }
}
```

```
public class Bucket {
//@ invariant elements.length == locks.length;
Object[] elements;
Lock[] locks;
AtomicInteger nrElements = new AtomicInteger();

public Bucket(int n) {
    elements = new Object[n];  locks = new Lock[n];
    for (int i = 0; i < n; i++) {
       elements[i] = new Object();  locks[i] = new ReentrantLock(); }

    nrElements.set(0);}

public int size() {return nrElements.get(); }

//@ requires i < elements.length;
public void add(int i, Object o) {
    locks[i].lock(); elements[i] = o; locks[i].unlock();
    nrElements.incrementAndGet(); }

//@ requires i < elements.length;
public void remove(int i) {
    locks[i].lock(); elements[i] = null; locks[i].unlock();
    nrElements.decrementAndGet();   }}
```


QUESTION 3

Where is the performance bottleneck in this code?

a. Too many locks
b. Locks maintained in an array
c. Locks protect too much data
d. nrElements accessed by all mutating operations

Note: try-catch forgotten on purpose, to save space

QUESTION 4

Thread 1:

```
while (true) {
    x = x + 1;
}
```

Thread 2:

```
print("YEP");
```

How to make sure this always prints "YEP"?

a. This requires a strongly fair scheduler, a weakly fair scheduler will not be sufficient
b. Yes
c. This requires a weakly fair scheduler
d. Only if x is volatile

QUESTION 5

Dining Philosphers

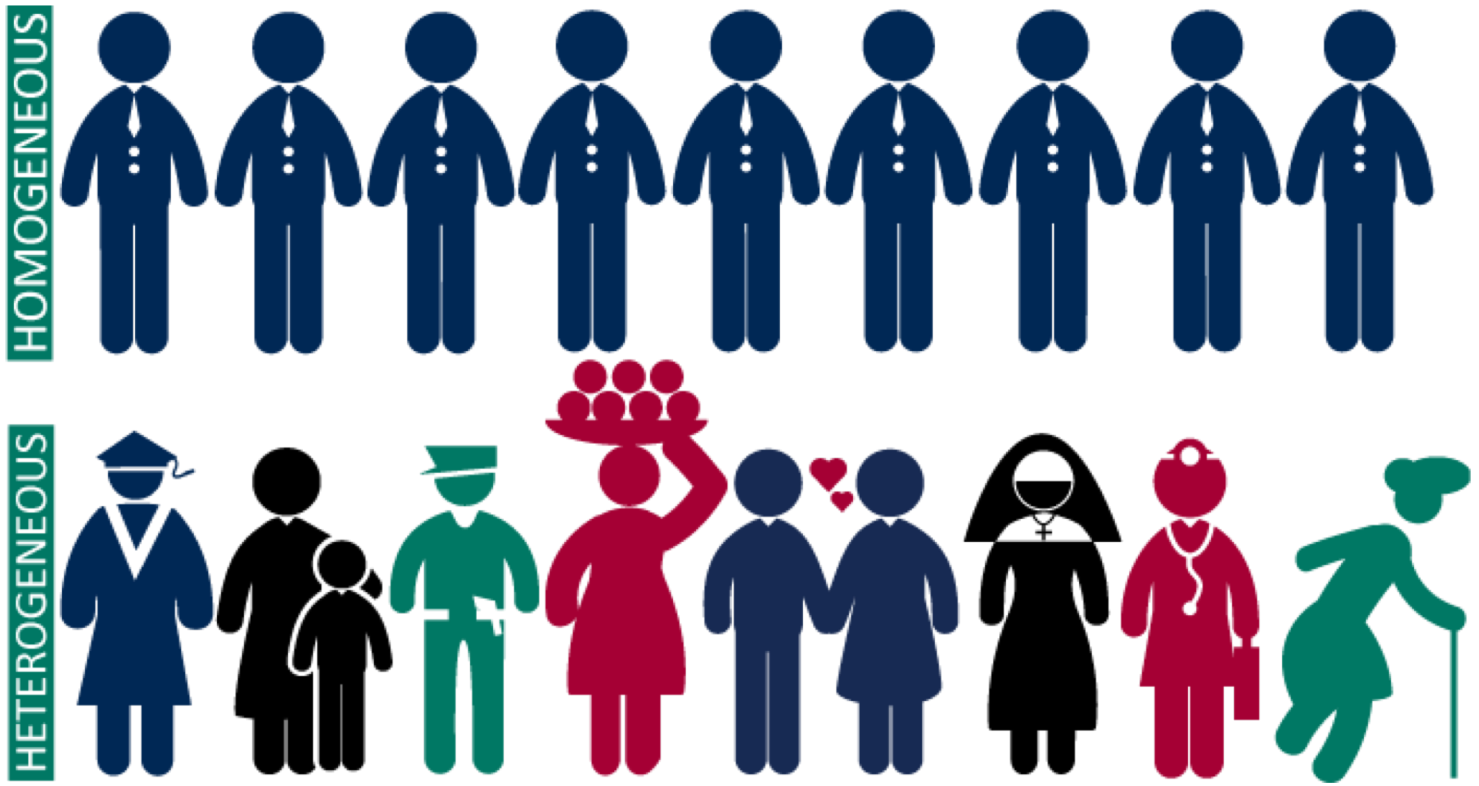
- Alternate in thinking and eating
- Philosopher needs two forks to eat

What can be said about this solution:

- Pick up left fork
- Try to get right fork
- If it fails, put left fork down
- Repeat

a. Deadlock
b. Starvation
c. Livelock
d. Problem solved

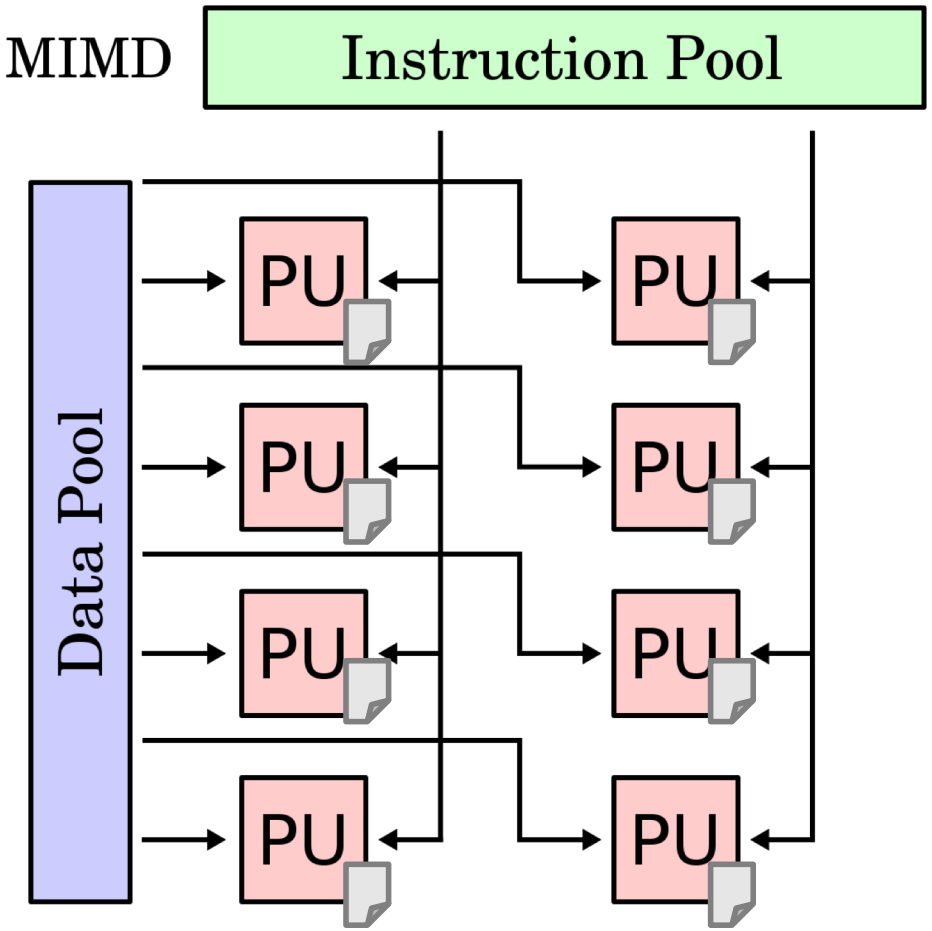
HETEROGENEOUS VS HOMOGENEOUS THREADING



HETEROGENEOUS THREADING

- Every thread executes its own program
- Threads share data
- Efficiency achieved by smart division of tasks

MIMD: MULTIPLE INSTRUCTION, MULTIPLE DATA

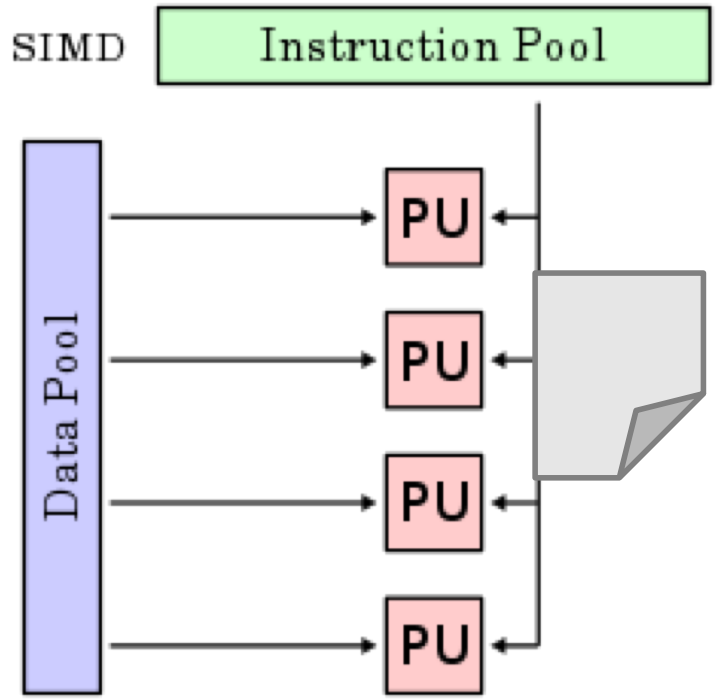


Architecture

HOMOGENEOUS THREADING

- Each thread executes same sequence of instructions
- Threads share data
- Typically, each thread accesses it own part of the data (otherwise: data race)
- Efficiency achieved by parallel execution of same job

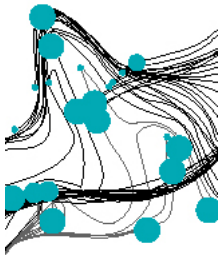
SIMD: SINGLE INSTRUCTION, MULTIPLE DATA



Architecture

- Many threads work on a vector, each on a different element
- They all execute the same instruction
- Hardware automatically handles divergence

- In fact: long-known model: vector machine
- One thread issues parallel vector instructions
- Fortran (still used in physics)



DEPENDENCES

INDEPENDENCE

```
for (int i = 0; i < a.length; i++) {
        a[i] = i;
}
```

Unrolling of the loop

```
a[0] = 0;

a[1] = 1;

a[2] = 2;

a[3] = 3;

…
```

- Every loop iteration independent of other iterations
- Order can be changed freely
- Parallellisation possible

LOOP-CARRIED FORWARD DEPENDENCE

```
for (int i = 0; i < a.length; i++) {
        a[i] = i + 3;
        if (i > 0) {
            c[i] = a[i - 1] + 2*i;
        }
}
```

- Second instruction in iteration j depends on first instruction in iteration j – 1 (write-before-use)
- Instructions can be re-ordered as long as assignment a[j – 1] always happens before assignment c[j]

Loop unrolling

a[0] = 0 + 3 = 3;

a[1] = 1 + 3 = 4;
c[1] = a[0] + 2*1 = 5;

a[2] = 2 + 3 = 5;
c[2] = a[1] + 2*2 = 8;

a[3] = 3 + 3 = 6;
c[3] = a[2] + 2*3 = 11;

…

POSSIBLE PARALLELLISATION

```
parallel {
        a[i] = i + 3;
}
synch
parallel {
        if (i > 0) {
            c[i] = a[i - 1] + 2*i;
        }
}
```

Possible behaviour

a[3] = 3 + 3 = 6;
a[0] = 0 + 3 = 3;
a[2] = 2 + 3 = 5;
a[1] = 1 + 3 = 4;
…

Synch

c[3] = a[2] + 2*3 = 11;
c[1] = a[0] + 2*1 = 5;
c[2] = a[1] + 2*2 = 8;

…

LOOP-CARRIED BACKWARD DEPENDENCE

```
for (int i = 0; i < a.length; i++) {
        a[i] = i + 3;
        if (i < a.length - 1) {
            c[i] = a[i + 1] + 2;
        }
}
```

- First instruction in iteration j + 1 depends on second instruction in iteration j (use-before-write)
- Could we parallellise this?

Loop unrolling

```
a[0] = 0 + 3 = 3;
c[0] = a[1] + 2;

a[1] = 1 + 3 = 4;
c[1] = a[2] + 2;

a[2] = 2 + 3 = 5;
c[2] = a[3] + 2;

a[3] = 3 + 3 = 6;
c[3] = a[4] + 2;

…
```

PARALLELLISATION BY COMPILER DIRECTIVES

OPENMP APPROACH

- Write a sequential program
- Make sure the program is correct
- Add compiler directives to give parallellisation hints
- Measure if performance improved by parallellisation hints
- If appropriate, repeat the process

```
# pragma omp parallel
    structured code block
```

This block may be parallellised

Structured parallel programming

PARALLELLISATION OF FOR LOOPS

Matrix-vector multiplication

With known number of iterations

```
void mxv(int m, int n, double* restrict a,
         double* restrict b, double* restrict c) {
  int i, j;
  for (i=0; i<m; i++) {
    a[i] = 0.0;
    for (j=0; j<n; j++)
      a[i] += b[i*n+j]*c[j];
  }
}
```

How can we parallellise this?

PARALLELLISATION OF FOR LOOPS

Matrix-vector multiplication

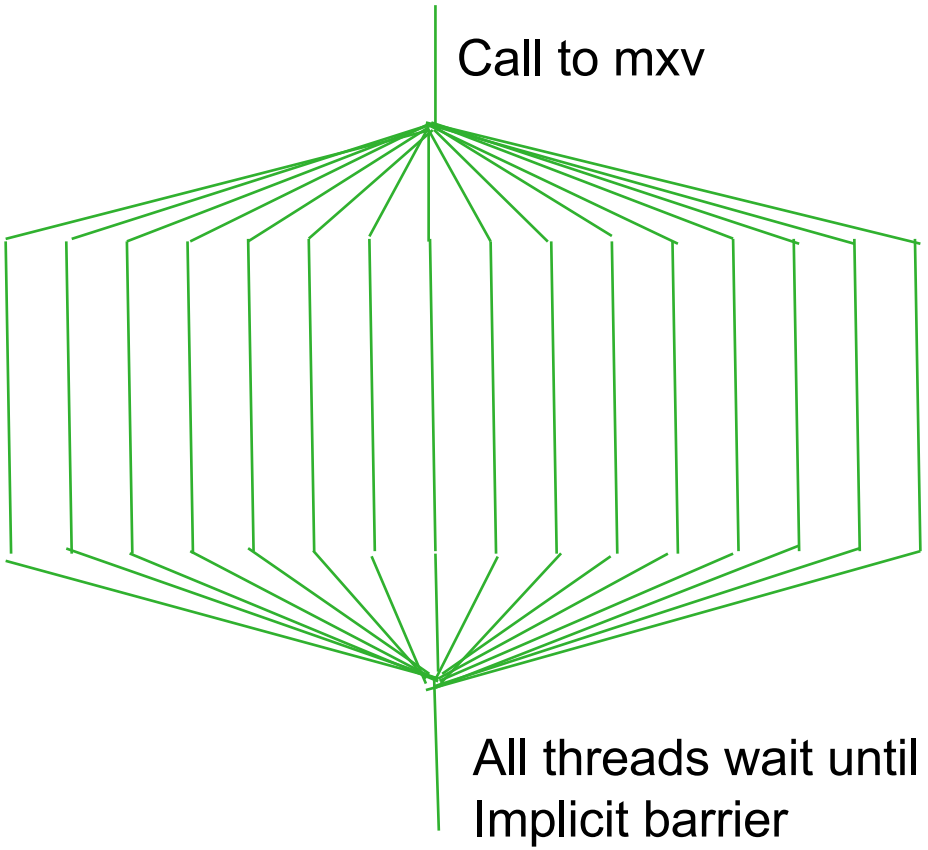
Parallellisation of outer for loop

```
void mxv(int m, int n, double* restrict a,
         double* restrict b, double* restrict c) {
  int i, j;
  #pragma omp parallel for default(none) shared(m,n,a,b,c) private(i,j)
  for (i=0; i<m; i++) {
    a[i] = 0.0;
    for (j=0; j<n; j++)
      a[i] += b[i*n+j]*c[j];
  } /*-- End of parallel region --*/
}
```

- Actually two pragmas
  - omp parallel
  - omp for
- Distinction between shared and private variables
- Default (none): no default assumptions about sharing/private

WHAT HAPPENS?

- Nesting of parallel is possible, but not all OpenMP APIs support it
- But what would this mean for performance?



m threads executing

```
a[i] = 0.0;
for (j=0; j<n; j++)
    a[i] += b[i*n+j]*c[j];
```

where i is current thread id

All threads wait until the others are done
Implicit barrier
Sequential continuation

SEQUENTIAL VS PARALLEL EXECUTION

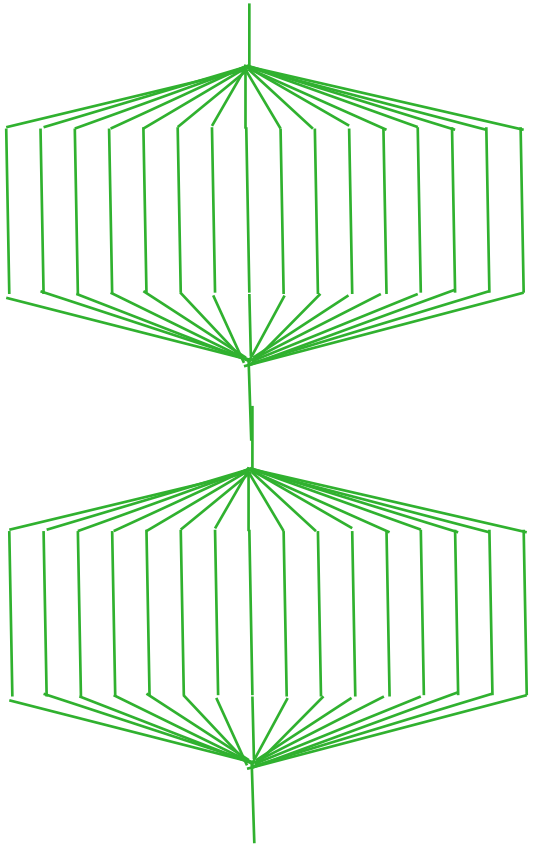
- Basic principle: program remains an executable sequential program
- Compiler directives can be ignored
- Some API calls such as omp_get_thread_num() violate this principle

- Solution: conditional compilation

```
#ifdef OPENMP
   #include <omp.h>
#else
   #define omp_get_thread_num() 0
#endif
int TID = omp_get_thread_num();
```

MULTIPLE LOOPS

```
#pragma omp parallel shared(n,a,b) private(i)
{
   #pragma omp for
   for (i=0; i<n; i++) {
      a[i] = i;
   }
   #pragma omp for
   for (i=0; i<n; i++) {
      b[i]=2*a[i];
   }
} /*-- End of parallel region --*/
```

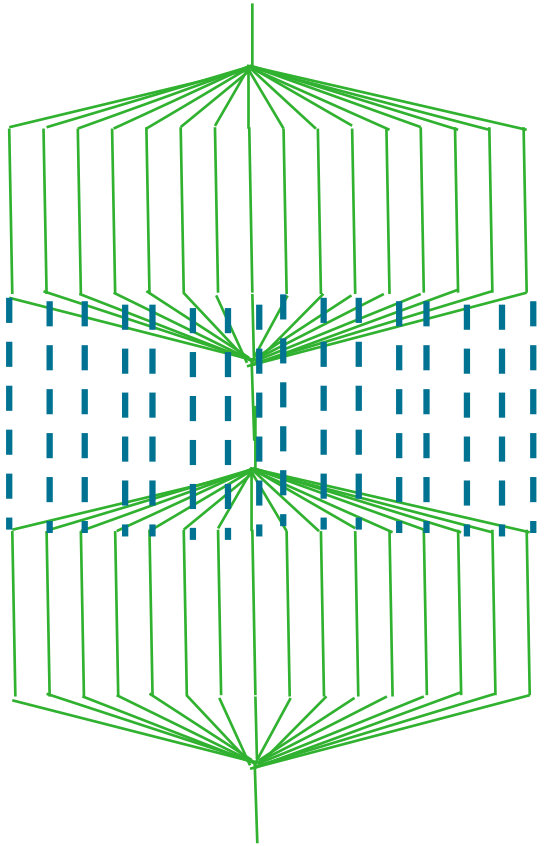


Could this be improved?

NOWAIT

Will this work?

```
#pragma omp parallel shared(n,a,b) private(i)
{
   #pragma omp for nowait
   for (i=0; i<n; i++) {
      a[i] = i;
   }
   #pragma omp for
   for (i=0; i<n; i++) {
      b[i]=2*a[i];
   }
} /*-- End of parallel region --*/
```

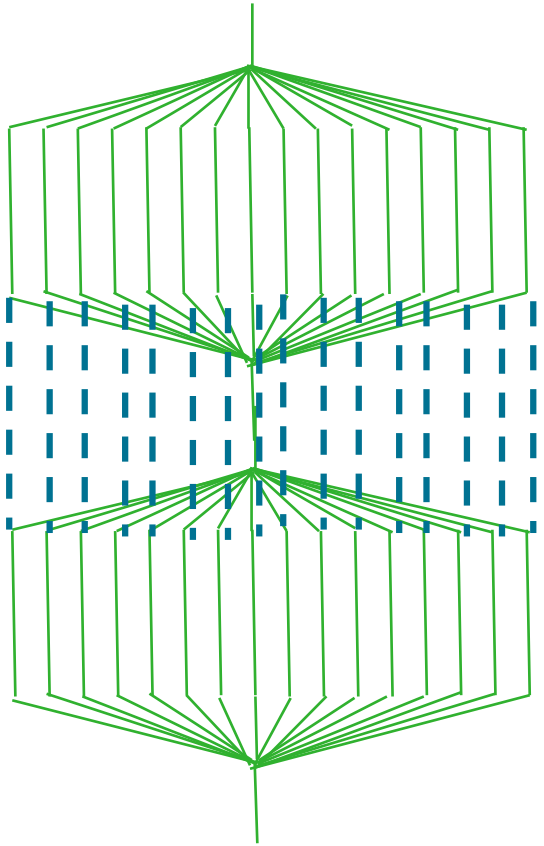


No, distribution over threads might be different!

NOWAIT

Static schedule:
Fix distribution of iterations over threads

```
#pragma omp parallel shared(n,a,b) private(i)
{
   #pragma omp for nowait schedule static
   for (i=0; i<n; i++) {
      a[i] = i;
   }
   #pragma omp for
   for (i=0; i<n; i++) {
      b[i]=2*a[i];
   }
} /*-- End of parallel region --*/
```



Now it works!

MORE PARALLEL EXECUTION

```
#pragma omp parallel
{
   #pragma omp sections
   {
      #pragma omp section
      (void) funcA();

      #pragma omp section
      (void) funcB();
   } /*-- End of sections block --*/
} /*-- End of parallel region --*/
```

- Execute funcA() and funcB() in parallel
- Note: now we are actually back at the heterogeneous execution model

MORE PARALLELLISATION?

Could we also parallellise the inner loop?

```
void mxv(int m, int n, double* restrict a,
         double* restrict b, double* restrict c) {
  int i, j;
  #pragma omp parallel for default(none) shared(m,n,a,b,c) private(i,j)
  for (i=0; i<m; i++) {
    a[i] = 0.0;
    for (j=0; j<n; j++)
      a[i] += b[i*n+j]*c[j];
  } /*-- End of parallel region --*/
}
```

The problem
All iterations access and update variable a[i]

REDUCTION LOOPS: SMALL EXAMPLE

Can we also parallellise this fragment?

```
sum=0;
for  (i=0;  i<n;  i++) {
   sum  +=  a[i];
}
```

Hint:
Every loop iteration has two basic steps:

- Read a[i]
- Write to shared variable sum

USING CRITICAL REGION

First compute locally
Then add up all the local results

```
#pragma omp parallel shared(n, a, sum) private(sumLocal)
{
   sumLocal=0;
   #pragma omp for
      for  (i=0;  i<n;  i++) {
         sumLocal  +=  a[i];
      }
   # pragma omp critical (update_sum)
      sum += sumLocal;
} /*-- End of parallel region --*
```

This is a very common pattern
Would be nice to be able to do this without rewriting the code

REDUCTION

```
#pragma omp parallel for default(none) shared(n,a) reduction(+:sum)
for (i=0; i<n; i++)
   sum += a[i];
/*-- End of parallel reduction --*/
```

- Roughly equivalent in behaviour to previous example
- But compiler can decide on more efficient implementations, for example using tree structure (see exercise)

Operator	Initialization value
+	0
*	1
-	0
&	~0 (= true)
|	0
^	0
&&	1
\|\| | 0

SIMD INSTRUCTION

SIMD support from OpenMP 4.0 with simd and for simd annotations where loop body executes in lock-step fashion.

```
#pragma omp parallel for simd  simdlen(M)
 for(int i =0;i<N;i++) {
     c[ i ]=a[i] +2;
   --------------------
     d[i] = c[i] *b[i];
 }
```

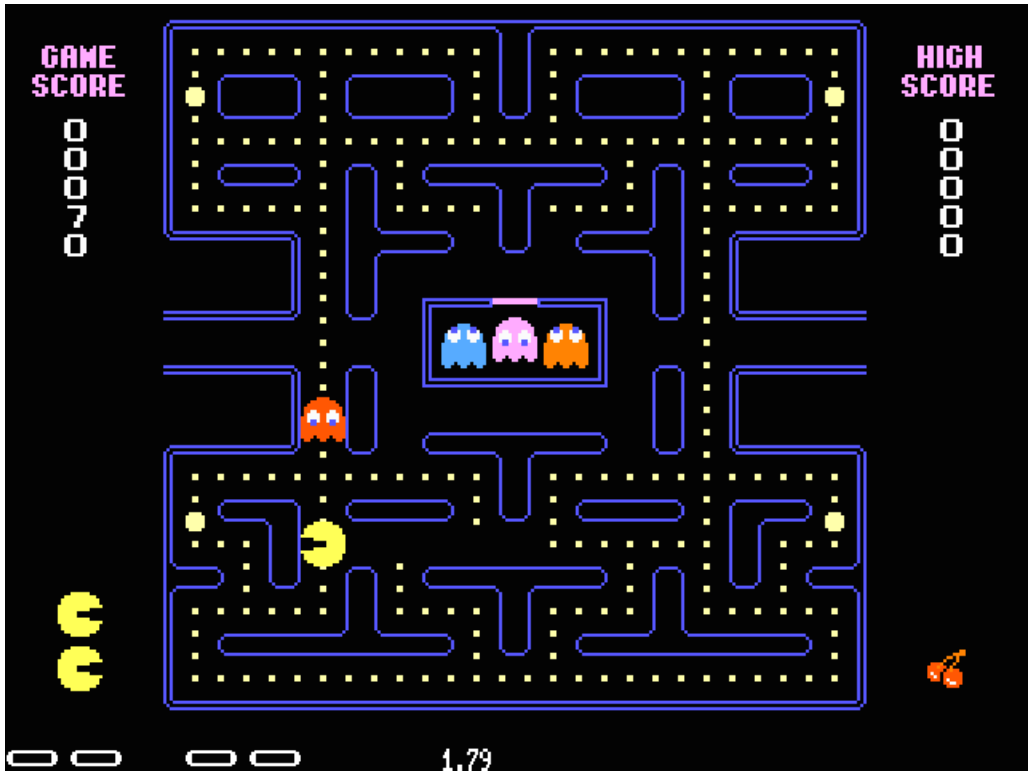
```
#pragma omp simd  simdlen(M)
 for(int i =0;i<N;i++) {
     c[ i ]=a[ i ] +2;
   --------------------
     d[i] = c[ i+M ] *b[i];
 }
```

THERE IS MUCH MORE

- Barrier (add explicit synchronisation)
- Single
- Other scheduling mechanisms
- Atomic
- Locks
- …

Effective use of OpenMP might require to write program in different way

GPU PROGRAMMING



GPU ARCHITECTURE

- Wikipedia:

  A graphics processing unit (GPU),

  also occasionally called visual processing unit (VPU),

  is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the building of images in a frame buffer intended for output to a display.

- SIMD architecture: built-in support for homogeneous threading
- Also useful for general purpose applications

GPU PROGRAMMING MODELS

- Cuda
  - NVIDIA-only
  - First
  - Widely-used
- OpenCL
  - Platform-independent
  - Can even run on CPU
  - Gaining interest

Essentially: an extended subset of C

MAIN CHARACTERISTICS GPU

- Architecture
  - Hundreds/thousands of slim cores
  - Homogeneous
- Memory
  - Complex hierarchy
  - Both shared and per-core
- Programming
  - Off-load model from CPU to GPU
  - Many fine-grained symmetrical threads
  - Hardware-specific scheduler

EXAMPLE: ADDITION OF TWO VECTORS

Sequential program:

```
void vector_add(int size, float* a, float* b, float* c {
    for(int index = 0; index < size; index++) {
        c[index] = a[index] + b[index];
    }
}
```

VECTOR ADDITION AS OPENCL KERNEL

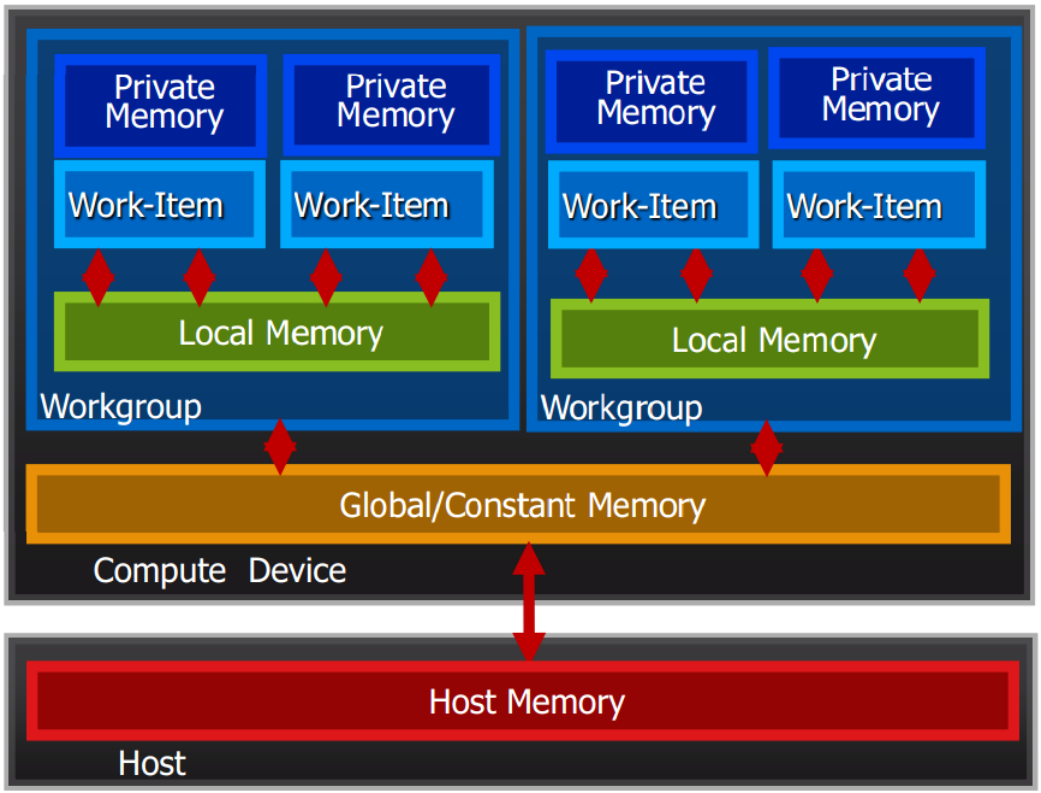
```
__kernel void vectorAdd(__global float* a,
                        __global float* b,
                        __global float* c) {
    int index = get_global_id(0);
    c[index] = a[index] + b[index];
}
```

__global
Where are the arrays stored

MULTIPLE MEMORY SCOPES

Beware for terminology:
- Local memory is shared by multiple work items (threads)
- Private memory is private to a single thread

- Per-thread private memory
- Per-workgroup shared memory
  - Low latency
- Global device memory (and constant memory)
  - Slower access
  - Can be accessed by any thread in any workgroup



Explicit copying between local and global memory

HIERARCHY OF CONCURRENT THREADS

- Parallel kernels composed of many threads
  - All threads execute the same sequential program
  - Called the kernel
- Threads (work items) are grouped into thread blocks (working group)
  - Threads in the same block can cooperate
  - Threads in different blocks cannot!

Thread identifiers typically determine which data is accessed

- Each thread has:
  - Local identifier: thread number in thread block
  - Global identifier: thread number in kernel

    Derived from local identifier and working group identifier

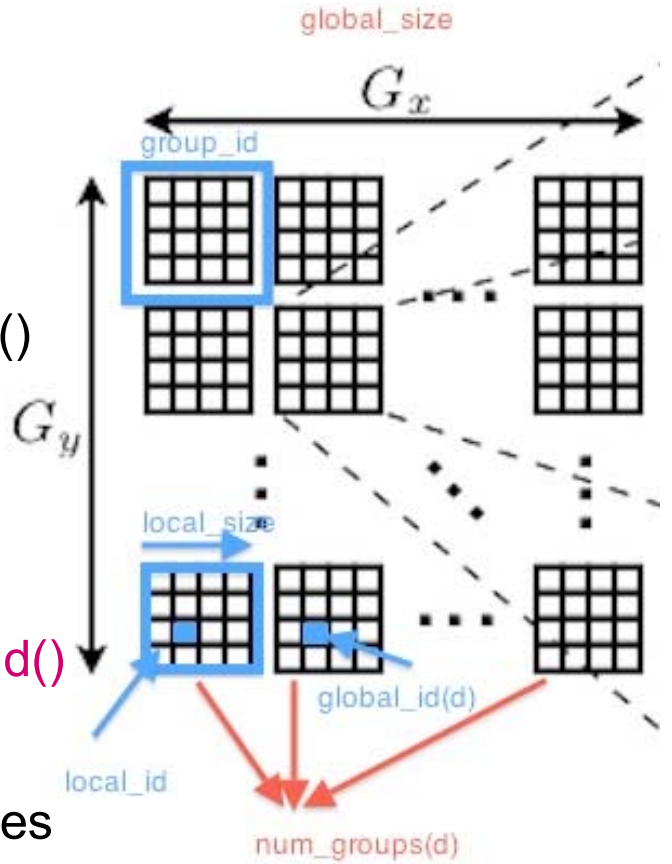
SOME USEFUL API FUNCTIONS

- Number of work groups: get_num_groups()
- Workgroup size: get_local_size()
- Workgroup identity: get_group_id()
- Thread identity in the workgroup: get_local_id()
- Global thread identity: get_global_id()

get_global_id() ==

get_local_id() + get_local_size() * get_group_id()

Also support for multi-dimensional thread identities



WHEN IS THIS SUITABLE?

CPU must be good at everything, parallel or not

- Minimize latency experienced by 1 thread
- Big on-chip caches
- Sophisticated control logic

GPU assumes work load is highly parallel

- Maximize throughput of all threads
- High number of parallel threads
- Multithreading can hide latency => skip the big caches
- Share control logic across many threads

Heterogeneous computing

- Part of work on CPU, part on GPU
- Find optimal balance between the two

HOST CODE: CPU

Host dispatches kernels to devices

Needs the following data:

- Device: which device will the code be executed on
- Kernel: all known programs that can run on the device
- Program: which program will be running on the device
- Command queue: where the program is send to the device
- Context: additional info needed by the device (workgroup size etc.)

Requires a lot of boilerplate code

HOST CODE INGREDIENTS

```
int main() {
    // allocate and initialize host (CPU) memory
    float *h_A = ..., *h_B = ...;
    // allocate device (GPU) memory
    cl_mem d_A, d_B, d_C;
    d_A = clCreateBuffer(clctx, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, N *sizeof(float),
    h_A, NULL);
    d_B = clCreateBuffer(clctx, CL_MEM_READ_ONLY |  CL_MEM_COPY_HOST_PTR, N *sizeof(float),
    h_B, NULL);
    d_C = clCreateBuffer(clctx, CL_MEM_WRITE_ONLY, N *sizeof(float), NULL, NULL);
    clkern=clCreateKernel(clpgm, “vadd", NULL);
    clerr= clSetKernelArg(clkern, 0,sizeof(cl_mem), (void *) &d_A);
    clerr= clSetKernelArg(clkern, 1,sizeof(cl_mem), (void *) &d_B);
    clerr= clSetKernelArg(clkern, 2,sizeof(cl_mem), (vern, 2, NULL, Gsz,Bsz, 0, NULL, &event);
    clerr= clWaitForEvents(1, &event);
    clEnqueueReadBuffer(clcmdq, d_C, CL_TRUE, 0, N*sizeof(float), h_C, 0, NULL, NULL);
    clReleaseMemObject(d_A);
    clReleaseMemObject(d_B);
    clReleaseMemObject(d_C);
}
```

REASONING ABOUT KERNEL CODE

```
__kernel bla (__global float* a) {
    int tid = get_global_id(0);
    a[tid] = 0;
    a[(tid + 1) % get_global_size()] = 1;
}
```

What will happen here?

Data races should be avoided
Synchronisation needed
Solution: insert a barrier between the two assignments

SYNCHRONISATION WITHIN A KERNEL

- Barrier: all threads within a work group block until all threads have reached (the same) barrier
- This is the only moment where you can make an assumption about the state of another thread
- Barriers can be flagged with empty, local, global or local & global
  - Flag indicates which memory is synchronised when all threads reach the barrier

BARRIER DIVERGENCE

- All threads synchronise on the same barrier
- Example of possible barrier divergence:

```
if b
   BARRIER(…);
else
   BARRIER(…);
```

Only okay if
all threads satisfy
b or not b

BARRIER DIVERGENCE?

```
__kernel void foo() {
    int i_max = (tid==0 ? 4 : 1);
    int j_max = (tid==0 ? 1 : 4);
    for(int i = 0; i < i_max; i++) {
        for(int j = 0; j < j_max; j++) {
            barrier();
        }
    }
}
```

Yes

- tid == 0 executes 4 times the inner for-loop
- tid != 0 executes the inner for-loop once, and iterates 4 times
- Not the same barrier!

REASONING ABOUT KERNEL CODE

```
__kernel bla (__global float* a) {
    int tid = get_global_id(0);
    a[tid] = 0;
    BARRIER(CLK_GLOBAL_MEM_FENCE);
    assert a[(tid + 1) % get_global_size()] == 0;
}
```

What can be said about validity of the assertion?

Barrier exchange information about thread state at barrier

KERNEL PATTERN: VECTOR ADDITION

```
__kernel void vectorAdd(__global float* a,
                        __global float* b,
                        __global float* c) {
   int index = get_global_id(0);
   local float* A; local float* B; local float* C;
   A[tid] = a[tid];     // move from global to local memory
   B[tid] = b[tid];
   barrier(CLK_LOCAL_MEM_FENCE);     // synchronise
   C[index] = A[index] + B[index];              // compute
   barrier(CLK_LOCAL_MEM_FENCE);     // synchronise
   c[tid] = C[tid];      // move from local to global memory
}
```

Okay, normally the local computations would be more complex…

REDUCTION

Could we turn this into a kernel?

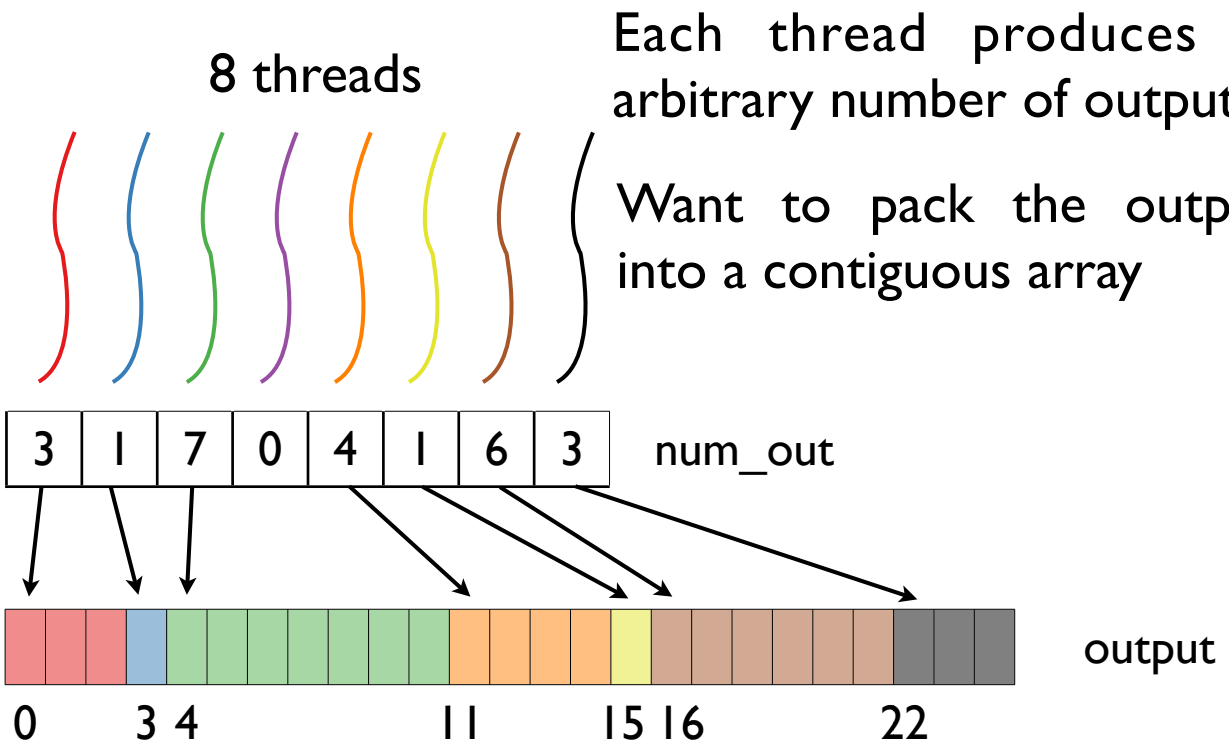
```
for (i=0; i<n; i++)
    sum += a[i];
```

We need a way to ensure that each thread does an atomic update

```
__kernel(__global float* a, __global int sum) {
    int tid = get_global_id(0);
    atomic_add(sum,a[tid]);
}
```

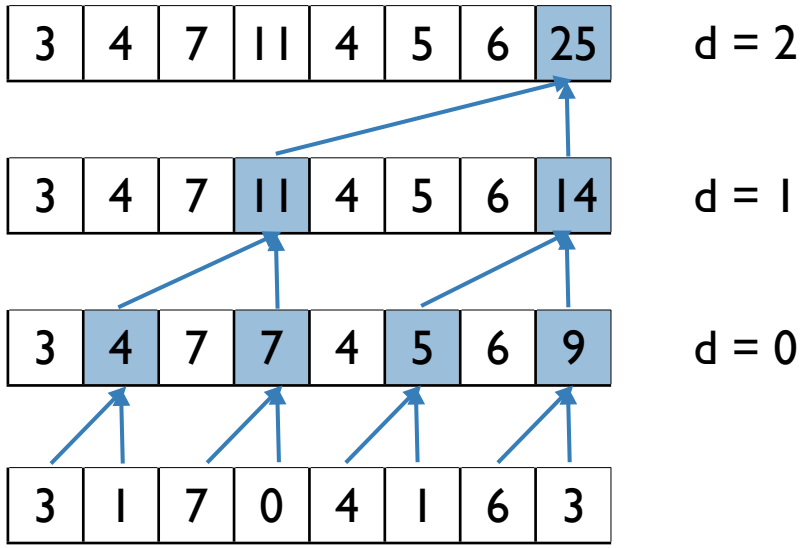
Performance: linear bottleneck
Exercise: a smarter way to do compute sum

PREFIX SUM OR SCAN



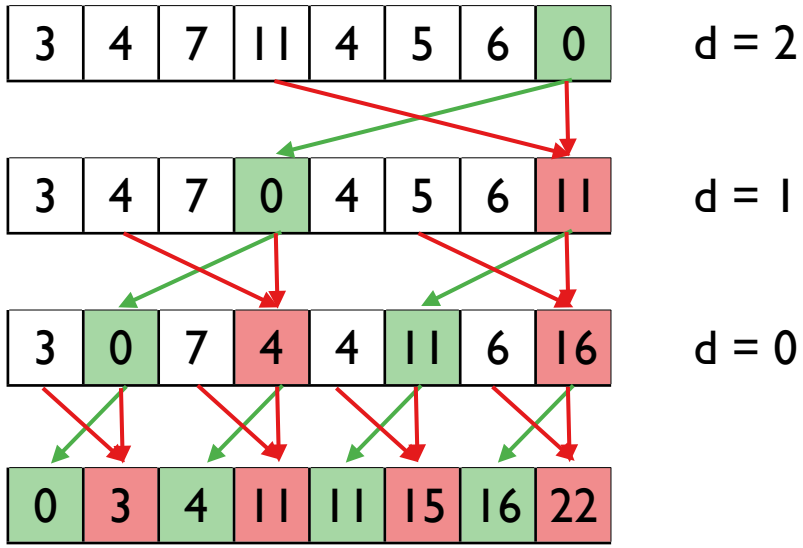
We look at one implementation
Many different versions exist

KERNEL IMPLEMENTATION: UPSWEEP



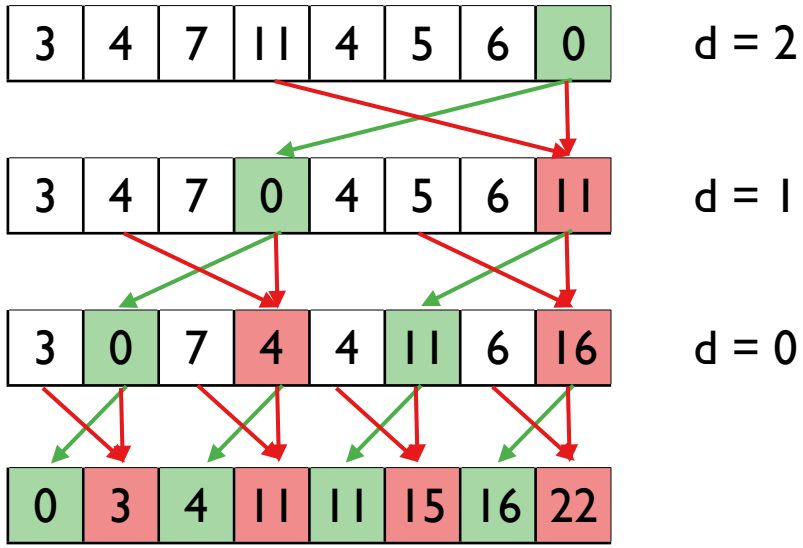
In every round from d = 0 to d = log n:
if (tid + 1) % 2^(d + 1) == 0
then
sum[tid] = sum[tid – 2^d] + sum[tid]

KERNEL IMPLEMENTATION: DOWNSWEEP



In every round from d = log n to d = 0:
if (tid + 1) % 2^(d + 1) == 0
then
    out[tid – 2^(d – 1)] = out[i]
    out[tid] = sum[tid – 2^(d – 1)] + out[i]

KERNEL IMPLEMENTATION: DOWNSWEEP



In every round from d = log n to d = 0:
if (tid + 1) % 2^(d + 1) == 0
then
    out[tid – 2^(d – 1)] = out[i]
    out[tid] = sum[tid – 2^(d – 1)] + out[i]

SUMMARY

- Heterogeneous and homogeneous threading
- Loop (in)dependences indicate possible parallellisations
- OpenMP: compiler directives
- OpenCL: vector programming with hardware support