Complexity vs Profilers

- *Asymptotic complexity*: what happens for large n?
  - Calculated by humans
  - *and/or* Estimated by empirical results
- *Real-world performance*: what happens for a specific n?
  - Estimated by humans
  - *and/or* Measured by machines

- Performance optimisation should be data-driven
  - Don’t start tweaking low-level code – go for algorithms
  - Measure ***before*** you start low-level optimisations
  - Measuring = profiling

Sampling vs Instrumentation

- Instrumentation
  - Modify generated code to have enter/exit routines
  - Record the entry/exit times
  - Sum the total time spent in each procedure

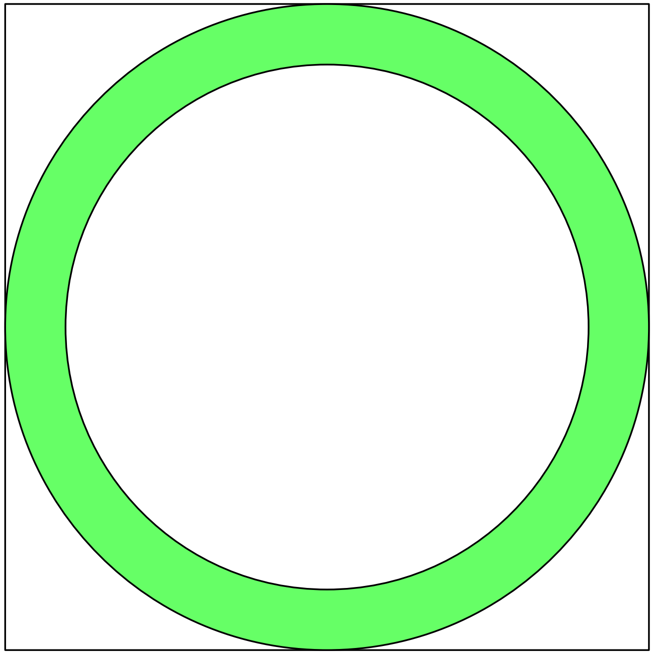
- Sampling
  - Periodically sample the call-stack
  - Record all the functions currently on the stack
  - Calculate percentage of time spent in each function

Recap: Estimate GPU execution time

- We have K threads/warp, N threads/local, M threads/global
  - Total threads = M
  - Warps / local = (N+K-1) div K
  - Warps / global = (M+K-1) div K
- Time of upper hierarchy is max of lower hierarchy
  - *any* active thread will keep a warp active
  - *any* active warp will keep a local group active
  - *any* active local group will keep a global group active
- Try to work out the worst-case thread execution time
  - ***Excellent*** : all threads follow the same instruction path
  - ***Good*** : all threads in a local group perform the same instructions
  - ***Ok*** : all threads in a warp take the same branches
  - ***Bad*** : each thread takes different path; divergence everywhere

Irregular iteration spaces

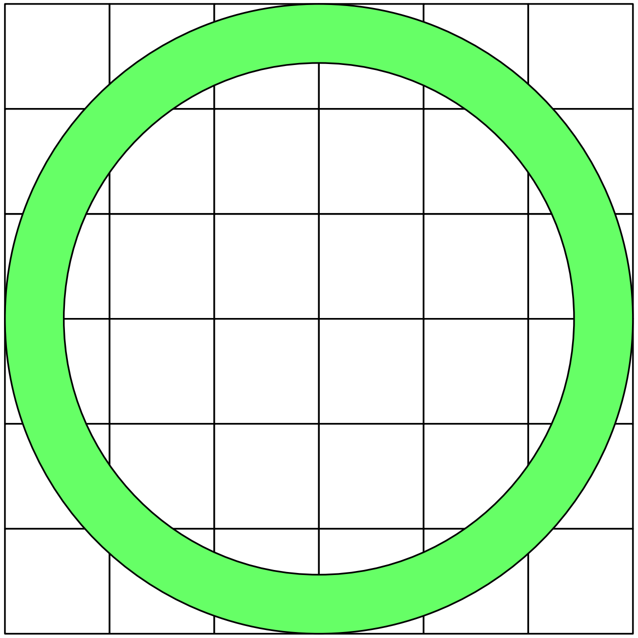
- Integration over a 2D torus
  - Outside torus integrand is zero

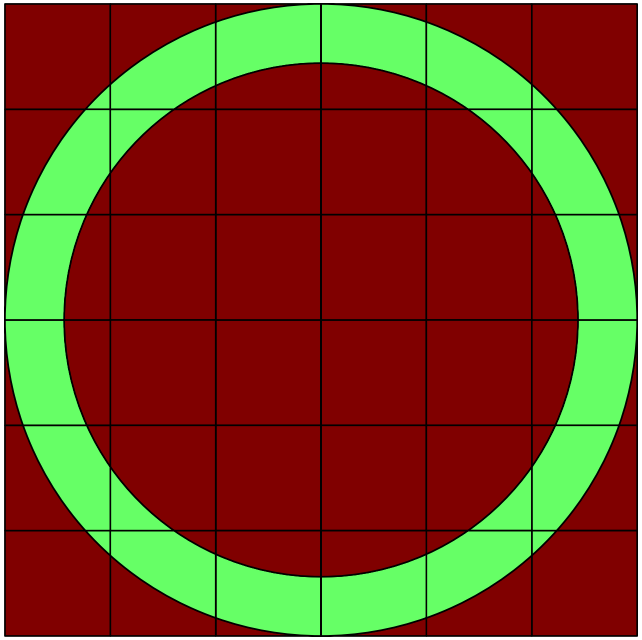


```
__kernel void Integrate(
   int ox,int oy, float dx,float sy,
   float inner_r, outer_r,
   float *result
){
   // Convert thread loc to point
   float x=(get_global_id(0)-ox)*sx;
   float y=(get_global_id(1)-oy)*sy;

   float integrand=0.0f;

   float r=sqrt(x*x + y*y);
   if( (inner_r<=r)||(r<=outer_r) ){
       // Horrible function with
       // hundreds of iterations
      integrand=CalculatePoint(x,y);
   }

   // Convert (x,y) to linear index
   int dst=get_global_dim(1)
       *get_global_id(0)
       +get_global_id(1);
   result[dst] = integrand;
}
```

Irregular iteration spaces

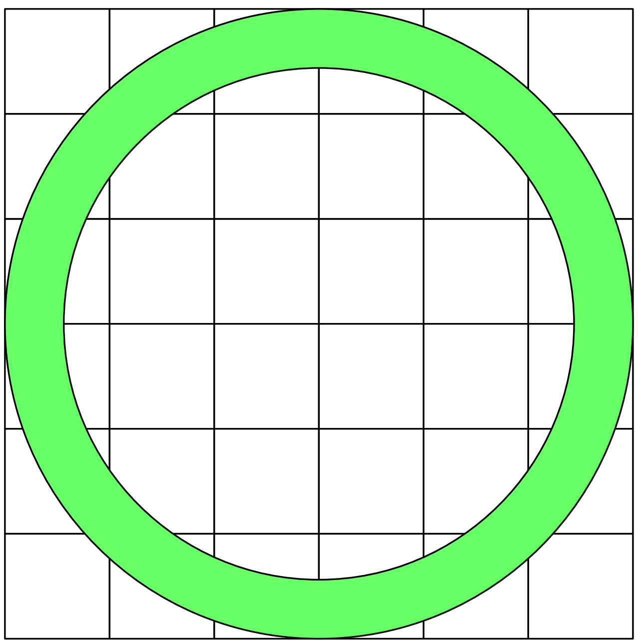
- Consider numeric integration over a 2D torus
  - Outside the torus the integrand is zero and contributes nothing
- Split the range of integration into blocks
  - Many threads are completely wasted – integrand is zero

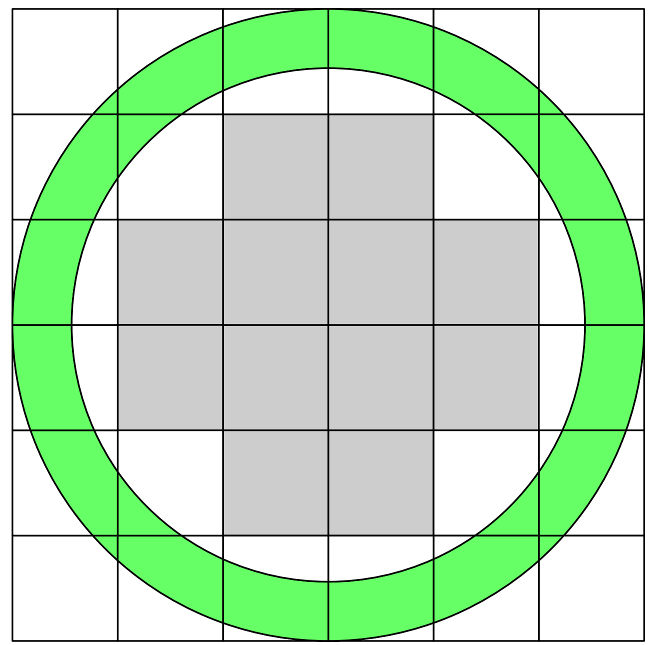




The joys of independent blocks

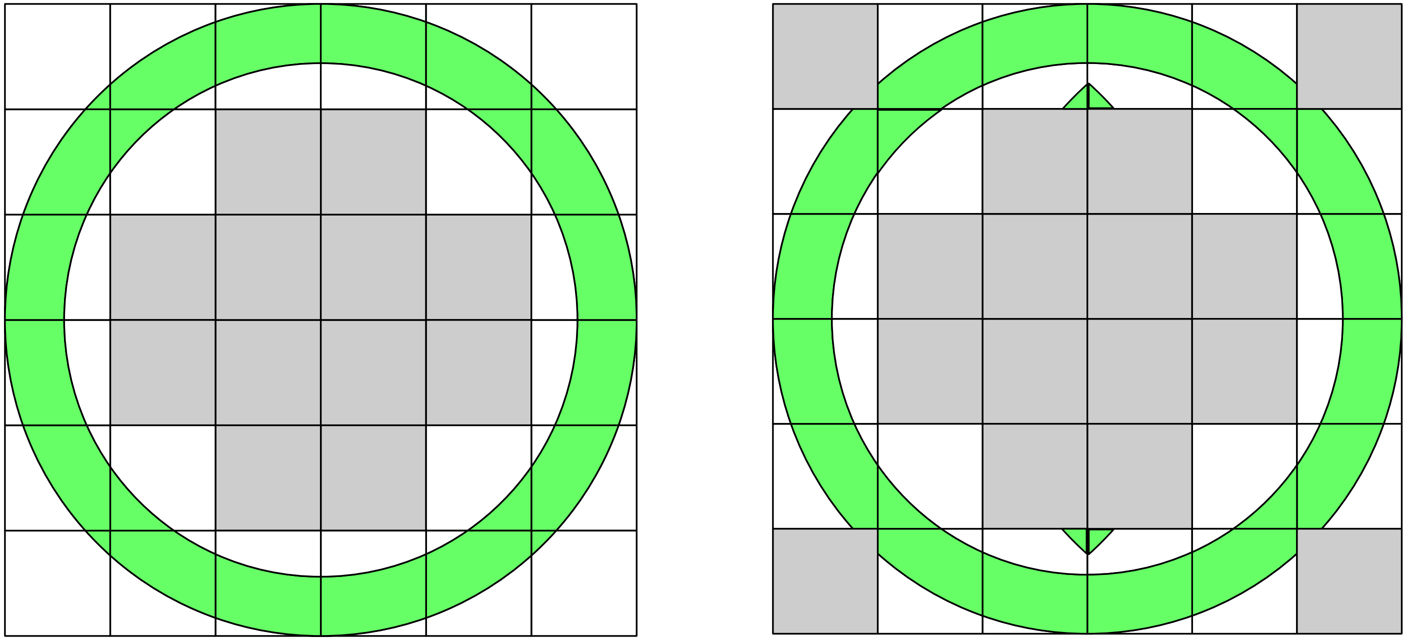
- If all threads in a local group don’t integrate, none of them do
- Local groups with few threads will exit with little overhead





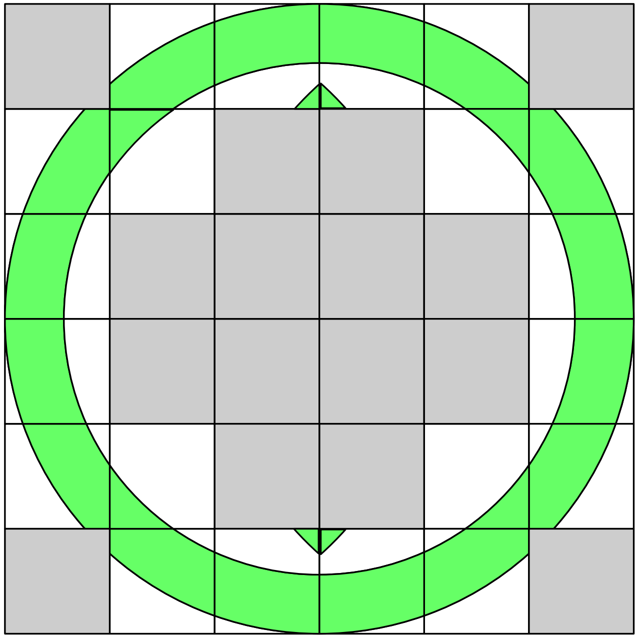
Manipulating iteration spaces

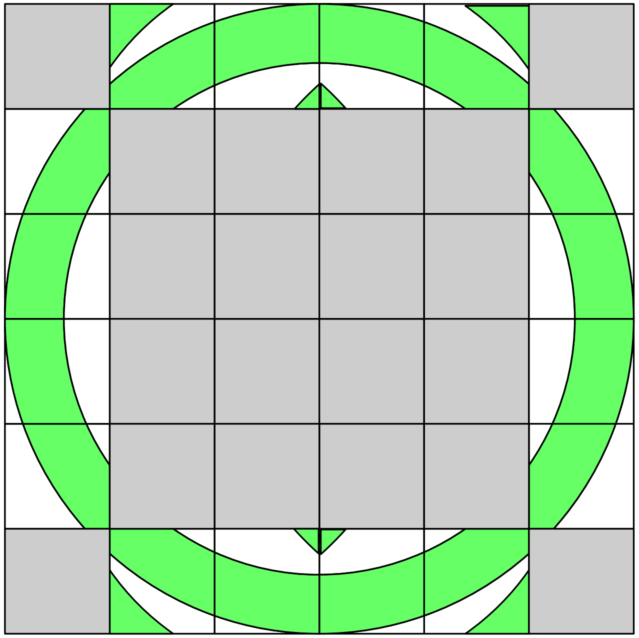
- Iteration spaces can be modified
  - Try to move lightly occupied regions into spare regions in other blocks



Manipulating iteration spaces

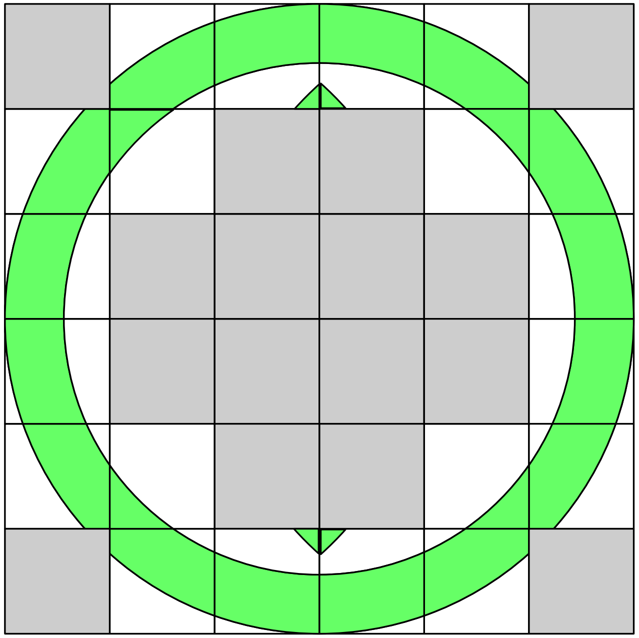
- Iteration spaces can be modified
  - Try to move lightly occupied regions into spare regions in other blocks
  - Empty local groups now take very little time

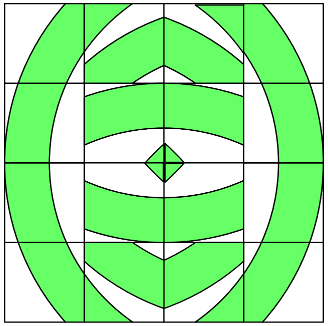




Manipulating iteration spaces

- Iteration spaces can be modified
  - Try to move lightly occupied regions into spare regions in other blocks
  - Empty local groups now take very little time
- Aggressive packing can reduce global size – reduce overhead





- How do you actually do the packing?
- What is the overhead of the mapping?

The global buffer anti-pattern

```
function MyFunction(nx, ny)
{
  buff1=zeros(nx,ny);

  % Setup the grid
  for x=1:nx
    for y=1:ny
      buff1(x,y)=SetupData(x,y);
    end
  end

  % Apply function to each element
  buff2=Transform(buff1);

  % Output data per point
  CalculateOutput(buff2);
}
```

```
__kernel void MyKernel()
{
  int x=get_id(0), y=get_id(1);
  int dst=x*tdim.y+y;
  __global int buff1[tdim.x*tdim.y];
  __global float buff2[tdim.x*tdim.y];

  buff1[dst]=SetupData(x,y);
  barrier(CL_GLOBAL_MEM_FENCE);

  buff2[dst]=Transform(buff1[dst]);
  barrier(CL_GLOBAL_MEM_FENCE);

  CalculateOutput(buff2[dst]);
  barrier(CL_GLOBAL_MEM_FENCE);
}
```

Pseudo-code only, not compilable

barrier : Synchronisation within the local work-group

The global buffer anti-pattern

- Beware global memory buffers
  1. I will initialise this array
  2. Then transform to next array
  3. Then get results from that array
- Bad for performance
  - Lots of pressure on off-chip RAM

```
__kernel void MyKernel()
{
  int x=get_id(0), y=get_id(1);
  int dst=x*tdim.y+y;
  __global int buff1[tdim.x*tdim.y];
  __global float buff2[tdim.x*tdim.y];

  buff1[dst]=SetupData(x,y);
  barrier(CL_GLOBAL_MEM_FENCE);

  buff2[dst]=Transform(buff1[dst]);
  barrier(CL_GLOBAL_MEM_FENCE);

  CalculateOutput(buff2[dst]);
  barrier(CL_GLOBAL_MEM_FENCE);
}
```

The global buffer anti-pattern

- Beware global memory buffers
  1. I will initialise this array
  2. Then transform to next array
  3. Then get results from that array
- – Bad for performance
  - – Lots of pressure on off-chip RAM
- – Try to move into shared memory
  - – Better bandwidth and latency

```
__kernel void MyKernel()
{
  int x=get_id(0), y=get_id(1);

  __local float buff[tdim.x];

  buff[x]=SetupData(x,y);
  barrier(CL_GLOBAL_MEM_FENCE);

  buff[x]=Transform(buff[x]);
  barrier(CL_GLOBAL_MEM_FENCE);

  CalculateOutput(buff[x]);
  barrier(CL_GLOBAL_MEM_FENCE);
}
```

The spurious array anti-pattern

- Beware global memory buffers
  1. I will initialise this array
  2. Then transform to next array
  3. Then get results from that array
- – Bad for performance
  - – Lots of pressure on off-chip RAM
- – Try to move into shared memory
  - – Better bandwidth and latency
- – Does data need to be in memory?
  - – Registers act as implicit arrays
  - – Thread location is the index

```
__kernel void MyKernel()
{
 int x=get_id(0), y=get_id(1);

  float buff;

  buff=SetupData(x,y);
  barrier(CL_GLOBAL_MEM_FENCE);

  buff=Transform(buff);
  barrier(CL_GLOBAL_MEM_FENCE);

  CalculateOutput(buff);
  barrier(CL_GLOBAL_MEM_FENCE);
}
```

False synchronisation anti-pattern

- Beware global memory buffers
  1. I will initialise this array
  2. Then transform to next array
  3. Then get results from that array
- Bad for performance
  - Lots of pressure on off-chip RAM
- Try to move into shared memory
  - Better bandwidth and latency
- Does data need to be in memory?
  - Registers act as implicit arrays
  - Thread location is the index
- Are the threads actually independent?
  - Remove false synchronisations

```
__kernel void MyKernel()
{
  int x=get_id(0), y=get_id(1);

  float buff=SetupData(x,y);

  buff=Transform(buff);

  CalculateOutput(buff);
}
```

Sharing memory in tasks and work-items

- What happens when parallel tasks share memory?
  - We can pass pointers to memory around (no-one stops us!)
  - Applies in both TBB tasks and OpenCL work-items: *all threads*

- Things we might want to happen:
  - Read-only : multiple tasks reading from constant data structure
  - Write-only : multiple tasks writing (but not reading) shared data
  - Read-write : tasks both reading and writing shared data

- Different tasks might use memory in different ways
  - e.g. one tasks reads, one task writes

Read-only access

- Simplest case: all tasks read from a shared data structure
- Absolutely fine, no consistency problems at all
- Except... structure must be written *before* tasks are launched

```cpp
void Worker(int n, const int *dst);

int main(int argc, char *argv[])
{
   unsigned n=100;
   std::vector<int> data(n,0);

   tbb::task_group group;
   group.run([&](){
      Worker(n, &data[0]);
   });

   for(unsigned i=0;i<n;i++)
       data[i]=g(i);

   group.wait();

   return 0;
}
```

```cpp
void Producer(int i, int *dst);
void Consumer(int i, int *dst);

int main(int argc, char *argv[])
{
   unsigned n=100;
   std::vector<int> data(n,0);

   tbb::task_group group;
   group.run([&](){  Producer(0, &data[0]);   });
   group.run([&](){  Producer(1, &data[n/2]); });

   DoOtherWork();

   group.run([&](){  Consumer(0, &data[0]);   });
   group.run([&](){  Consumer(1, &data[n/2]); });

   group.wait();

   CollectResults(&data[0]);

   return 0;
}
```

Be careful to launch your workers after creating their input, and to add a barrier (barrier or wait) between tasks creating output for consumption by other tasks.

Read-Write access

- Obviously a problem – this is what makes parallel code hard

- Tasks may execute according to any valid schedule
  - May execute sequentially even if there are multiple processors
  - May execute more tasks than physical processors

- Instructions of each tasks may be interleaved in any order
  - *Run all instructions of task A, then all instructions of B*
  - *Run the first five instructions of A, then one of B, then two of A...*

- Many of those instructions will be memory operations

Classic example: Counters

```
void TaskFunc(int *data)
{
    ...

    (*data) += 1;

    ...
}
```

```
void TaskFunc(int *data)
{
    ...

    int curr=*data;
    int next=curr+1;
    *data=next;

    ...
}
```

Classic example: Counters

```
void TaskFunc(int *data)              void TaskFunc(int *data)
{                                     {
    ...                                   ...
--------------------------------------------------------------------------
    int curr=*data;
--------------------------------------------------------------------------
                                          int curr=*data;
--------------------------------------------------------------------------
    int next=curr+1;                      int next=curr+1;
--------------------------------------------------------------------------
                                          *data=next;
--------------------------------------------------------------------------
    *data=next;
                                          ...
                                      }
    ...
}
```

Write-only access

- Multiple tasks will write to a particular memory location
  - Tasks will not read from location at all
  - Common pattern when searching for something
- No read-modify-write problems, as we don’t do a read first

- Seems fine – is it?
  - `char` : A char is (usually) the smallest addressable unit
  - `int` : Usually a register-width integer, atomic writes
  - `uint64_t`: A 64-bit integer – most machines are 64-bit?
- What about complex types?
  - `std::complex<double>` - Two 8-byte floating-point values
  - Moving a complex is often two or more instructions

`tbb::atomic<T>`

- Template class which makes an ***atomic*** version of type T
  - ***Atomicity***: an operation happens in a single indivisible step
  - Impossible to observe the state “half-way through”

- Can use `tbb::atomic` to hold things like counters
  - `tbb::atomic<int> x;` Integer counter we can share
  - `int v=x++;` Increments by one as atomic step

- To provide atomicity the operations on integer are limited
  - No atomic x<5, or x *= 5
  - Can only use fundamental integer types: T={`int`, `long`, ...}

Atomics beyond TBB

- C++11 includes built-in support for atomic operations
  - `std::atomic<T>` : very similar to `tbb::atomic<T>`
  - Also has free function to work with arbitrary variables:

```
template<class T>
T std::atomic_fetch_add (T* obj, T val);
```

  - Should generally stick with the class, it makes intentions clear

- OpenCL 1.1 supports atomic integers in local and global mem
  - `uint atomic_add (volatile __global uint *p, uint val);`
  - `uint atomic_add (volatile __local uint *p, uint val);`
  - Some platforms support atomic operations on floats and 64 bit integers
  - (No atomic operations for private memory – why?)

Unique ids with `tbb::atomic<int>`

- Common example is to acquire a unique integer id for *n* tasks
  - Each task receives a unique integer id
  - Each integer id is used exactly once

```
unsigned unique_id(unsigned *counter)
{
   return (*counter)++;
}


unsigned unique_id(tbb::atomic<unsigned> *counter)
{
   return (*counter)++;
}
```

Finding a shared minimum

- Need to use `fetch_and_store` to atomically exchange value
  - Atomically reads the current value while writing the new value

    `T tbb::atomic<T>::fetch_and_store(T value);`

```
void atomic_min(tbb::atomic<unsigned> *res, unsigned curr)
{
    while(true){
        unsigned prev=res->fetch_and_store(curr);
        if(prev>curr)
            break;
        curr=prev;
    }
}
```

  - OpenCL 1.1 has an `atomic_min` primitive, with HW support

`tbb::atomic<T *>`

- We often need to share "big" stuff
  - Tasks need to pass complicated data-structures of type X
- We can't use `tbb::atomic<X>` directly if X is "complex"
  - Cannot be implemented without using mutexes
  - TBB restricts you to using types which are fast
- Instead we can use a pointer to X to hold the object
  - Operations on *pointers* can be atomic

```
void atomic_max(
    tbb::atomic<std::string *> *res,
    std::string *curr
){
   while(true){
       std::string *prev=res->fetch_and_store(curr);
       if(prev==0)
           break;
       if( (*prev) < (*curr) ){
           delete prev;
           break;
       }
       curr=prev;
   }
}
```

```
void atomic_max(
    tbb::atomic<std::string *> *res,
    std::string *curr
){
  std::string value=*curr;

  while(true){
      std::string *prev=res->fetch_and_store(curr);
      if(prev==0)
         break;
      if( (*prev) < value ){
         delete prev;
         break;
      }
      curr=prev;
      value=*curr;
   }
}
```

Hidden dangers of `tbb::atomic`

- All operations on atomics are ***non-blocking***
  - Operations may be fairly slow, but can always complete
  - No task can block another task at an atomic instruction
- But atomic operations can be used to ***implement*** blocking
- So far our basic parallelism primitives have been “safe”
  - TBB allows shared read-write memory, but assumes you won’t treat it as atomic or consistent – very unlikely to be safe
  - run/wait parallelism creates a Directed Acyclic Graph
  - parallel_for is also safe, only creates spawn/sync dependencies
- Deadlock occurs where there is a ***circular dependency***
  - A depends on B; B depends on C; C depends on A
  - We can (accidentally or intentionally) make them using atomics

Atomics for reduction

- Atomics are very useful for fold or reduce operations
  - *Count the number of times event X occurred across all tasks*
  - *Track the total sum of Y across all tasks*
  - *Find the minimum value of Z across all tasks*
- For values $x_1..x_n$ calculate some property $f(x_1,f(x_2,f(x_3...x_n)))$
  - Function f(.) should be associative and commutative
  - **Associative** : f(a,f(b,c)) = f(f(a,b),c)
  - **Commutative** : f(a,b) = f(b,a)
  - Means that the result will be *independent of order of calculation*
- Useful reduction operators are often based on statistics
  - Count, Mean, Variance, Covariance, Min, Max
  - Histograms: arrays of `atomic<int>` ...

- There is an overhead associated with atomic operations
  - Try to reduce total number of operations per task
- Overhead increases when many CPUs work with same atomic
  - Limit the number of tasks sharing an atomic
  - Exploit the associative and commutative properties

```
void Worker(tbb::atomic<int> *prev, ...)
{
  if(SomeFunc()){
    Calculate(prev, ...);
  }else{
    // Only shared by my child tasks
    tbb::atomic<int> local;
    tbb::task_group group;

    for(unsigned i=0;i<4;i++){
      group.run([&](){ Worker(&local,...); });
    }

    group.wait();

    // Accumulate into parent's counter
    (*prev) += local;
  }
}
```

```
void Calculate(
  tbb::atomic<int> *dst, ...)
{
   int acc=0;
   for(int i=0;i<N;i++){
       // Accumulate locally
       // into non-atomic var
       acc += F(acc);
   }
   // Then one expensive add
   (*dst) += acc;
}
```