

CS 6023 - GPU Programming

Misc: Dynamic Parallelism, Thrust, OpenACC

09/11/2018

Setting and Agenda

- Dynamic parallelism
- Thrust - Example of using libraries
- OpenAcc

Dynamic Parallelism

- What kinds of programming patterns require dynamic parallelism?

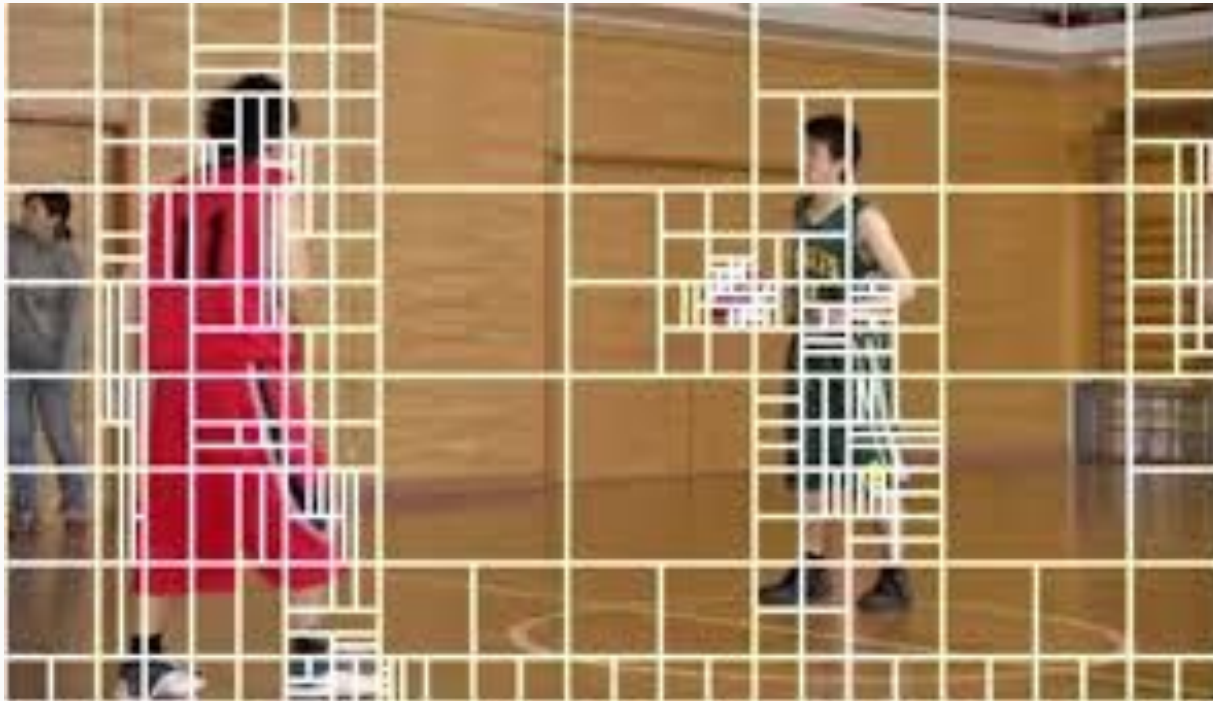
Dynamic Parallelism

- What kinds of programming patterns require dynamic parallelism?
 - Recursive
 - Irregular work structure
 - Reactive systems where work is uncovered at runtime

Dynamic Parallelism

- What kinds of programming patterns require dynamic parallelism?
 - Recursive - factorial
 - Irregular work structure - bezier curve, video encoding
 - Reactive systems where work is uncovered at runtime - object detection in deep learning

CU Tree example from video encoding



Simple example

```
for i = 1 to N
  for j = 1 to x[i]
    do_something(y[i, j])
```

- How do we implement this without any dynamic parallelism?

Simple example

```
for i = 1 to N
  for j = 1 to x[i]
    do_something(y[i, j])
```

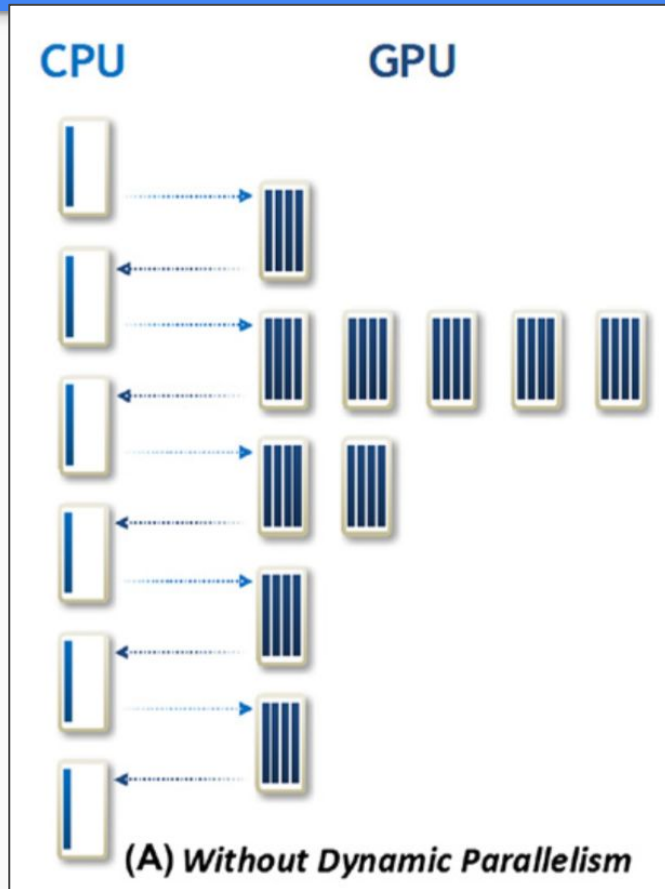
- How do we implement this without any dynamic parallelism?
 - Single kernel with device function call
 - Multiple kernels called by CPU

Simple example

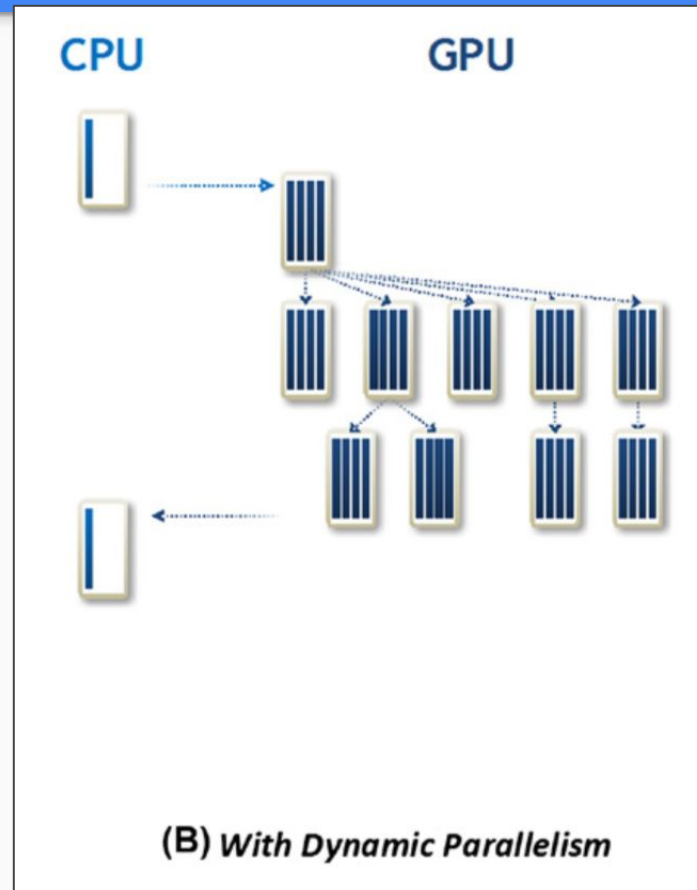
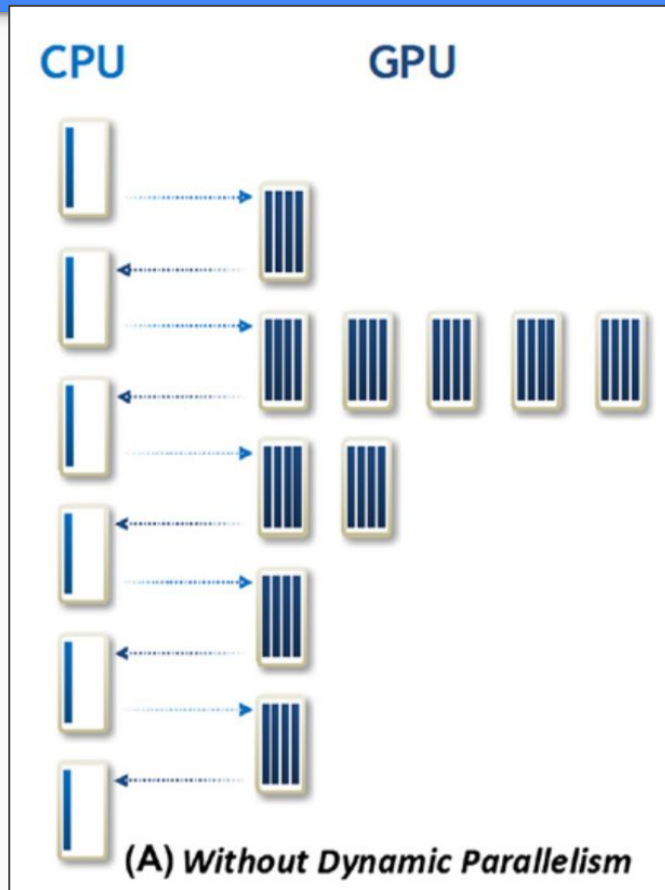
```
for i = 1 to N
  for j = 1 to x[i]
    do_something(y[i, j])
```

- How do we implement this without any dynamic parallelism?
 - Single kernel with device function call
 - Multiple kernels called by CPU
- What are the issues with these options?
 - Do not fully utilize parallelism or load balance
 - Does not support or optimize when two layers of recursion are allowed

With and without dynamic parallelism



With and without dynamic parallelism



Benefits of dynamic parallelism

Benefits of dynamic parallelism

- Makes the CPU - GPU partitioning simpler
- Simpler programming construct
- Load balancing and increasing occupancy
- Reduces memory copying (CPU to GPU and within GPU)

Same syntax

```
__global__ myFunc B(float *d) {  
    doWork(d);  
  
    K1 <<< ... >>> (d);  
    K2 <<< ... >>> (d);  
    K3 <<< ... >>> (d);  
  
    cudaDeviceSynchronize();  
  
    doMoreWork(d);  
}
```

How does sync work now?

```
__global__ myFunc B(float *d) {  
    doWork(d);  
  
    K1 <<< ... >>> (d);  
    K2 <<< ... >>> (d);  
    K3 <<< ... >>> (d);  
  
    cudaDeviceSynchronize();  
  
    doMoreWork(d);  
}
```

How does sync work now?

```
__global__ myFunc B(float *d) {  
    doWork(d);  
  
    K1 <<< ... >>> (d);  
    K2 <<< ... >>> (d);  
    K3 <<< ... >>> (d);  
  
    cudaDeviceSynchronize();  
  
    doMoreWork(d);  
}
```

- Sync works now in nested loops
- When CPU runs this function we have CPU -> K1, K2, K3 -> CPU sync
- When a kernel say KX runs this function, then we have an internal sync as KX -> KX1, KX2, KX3 -> KX

How does sync work now?

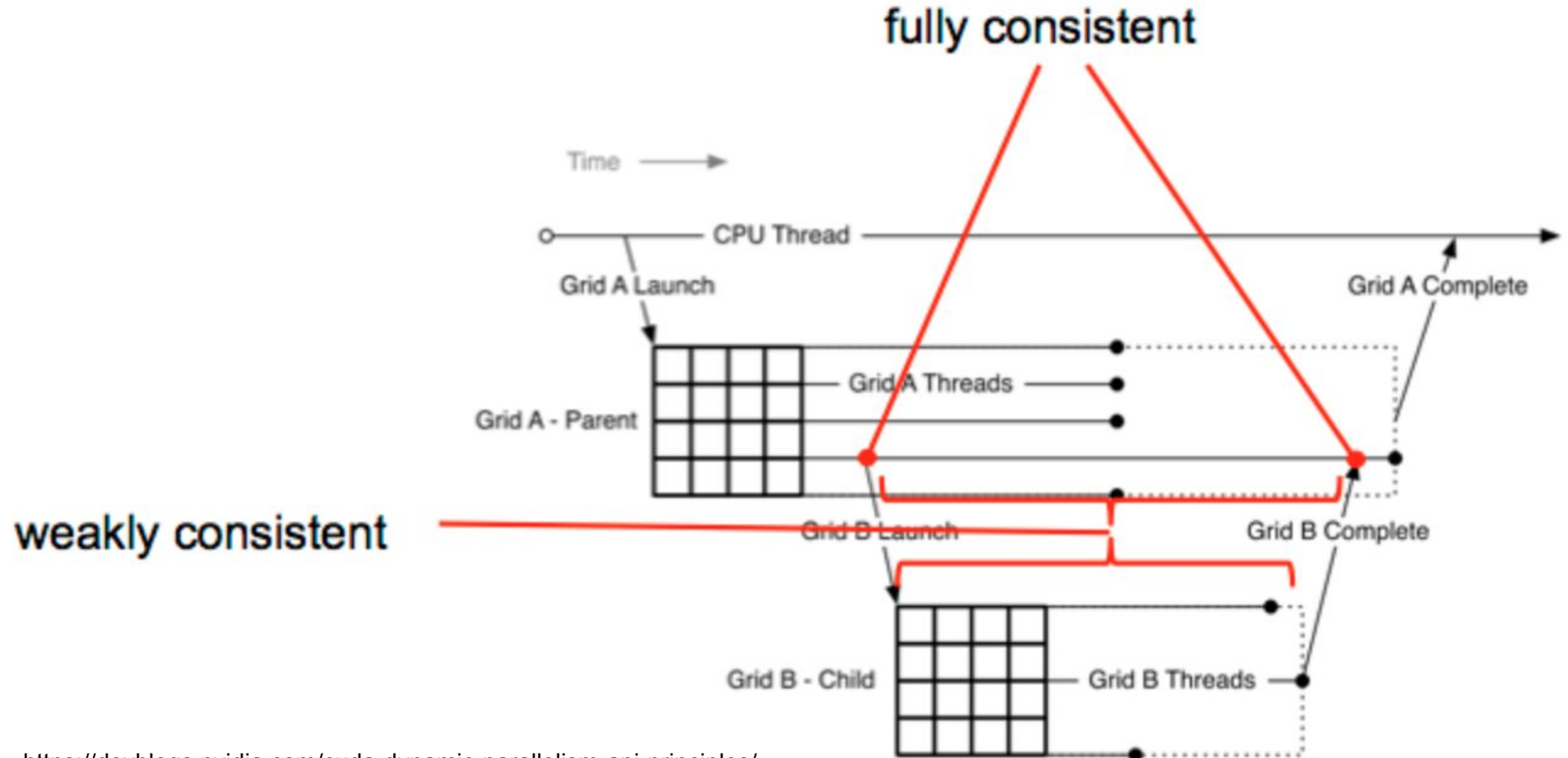
```
__global__ myFunc B(float *d) {  
    doWork(d);  
  
    K1 <<< ... >>> (d);  
    K2 <<< ... >>> (d);  
    K3 <<< ... >>> (d);  
  
    cudaDeviceSynchronize();  
  
    doMoreWork(d);  
}
```

- Sync works now in nested loops
- When CPU runs this function we have CPU -> K1, K2, K3 -> CPU sync
- When a kernel say KX runs this function, then we have an internal sync as KX -> KX1, KX2, KX3 -> KX
 - When in the GPU invocation `cudaDeviceSynchronornize()` is different from `__syncthreads()`

How do we deal with memory?

- Local and shared memory
 - Not shared between parent and child kernels
- Global memory
 - Two points of sync:
 - At the start of child launch, child sees the global memory as updated by parent before that
 - At the sync after a child completion, parent sees all updates made by child
- Constant memory - cannot be changed

How do we deal with memory?



Thrust

- Library for CUDA C/C++
- Think of it as a parallel to STL for C++ or BLAS for linear algebra
- Features include
 - Dynamic and templated data structures
 - Simple algorithms operating on these data structures
 - Encapsulation of CPU-GPU memory transfers

Sample program to sort arrays

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>
int main(void) {
    // generate 16M random numbers on the host
    thrust::host_vector<int> h_vec(1 << 24);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);
    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;
    // sort data on the device
    thrust::sort(d_vec.begin(), d_vec.end());
    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
    return 0;
}
```

Include libraries

Sample program to sort arrays

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>
int main(void) {
    // generate 16M random numbers on the host
    thrust::host_vector<int> h_vec(1 << 24);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);
    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;
    // sort data on the device
    thrust::sort(d_vec.begin(), d_vec.end());
    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
    return 0;
}
```

thrust:: namespace
Templated dynamic
types - containers

Sample program to sort arrays

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>
int main(void) {
    // generate 16M random numbers on the host
    thrust::host_vector<int> h_vec(1 << 24);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);
    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;
    // sort data on the device
    thrust::sort(d_vec.begin(), d_vec.end());
    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
    return 0;
}
```

Functions on
containers
Support for iterators

Sample program to sort arrays

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>
int main(void) {
    // generate 16M random numbers on the host
    thrust::host_vector<int> h_vec(1 << 24);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);
    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;
    // sort data on the device
    thrust::sort(d_vec.begin(), d_vec.end());
    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
    return 0;
}
```

Implicitly running
allocation, memcpy

Sample program to sort arrays

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>
int main(void) {
    // generate 16M random numbers on the host
    thrust::host_vector<int> h_vec(1 << 24);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);
    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;
    // sort data on the device
    thrust::sort(d_vec.begin(), d_vec.end());
    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
    return 0;
}
```

Implicitly taking care of
allocating and freeing
memory

When to use Thrust

- If your program requires careful allocation of resources and optimization, write CUDA native code
 - Kernel fusion, tiling, managing control divergence, coalesced memory accesses, ...
- However, if there are parts of the code that are standard (eg. sorting) and will not benefit from custom optimization, then use Thrust
- In general, libraries are ways for us to delegate our programming effort at the expense of control

OpenACC

- Short for Open Accelerator
- Intent is to have a **simple** way to **incrementally and transparently modify** existing sequential code so that it can be accelerated on **variety** of heterogeneous hardware targets
- Open source project with multiple companies including Cray and Nvidia and support for hardware across vendors: Nvidia, AMD, Intel, ...



```
#pragma acc data copyin(a,b) copyout(c)
{
    ...
    #pragma acc parallel
    {
        #pragma acc loop gang vector
        for (i = 0; i < n; ++i) {
            z[i] = x[i] + y[i];
            ...
        }
        ...
    }
}
```

Programmer specifies
directives for the compiler to
parallelise the code

```
#pragma acc data copyin(a,b) copyout(c)
{
    ...
    #pragma acc parallel
    {
        #pragma acc loop gang vector
        for (i = 0; i < n; ++i) {
            z[i] = x[i] + y[i];
            ...
        }
        ...
    }
}
```

Moving data

```
#pragma acc data copyin(a,b) copyout(c)
{
    ...
    #pragma acc parallel
    {
        #pragma acc loop gang vector
        for (i = 0; i < n; ++i) {
            z[i] = x[i] + y[i];
            ...
        }
        ...
    }
}
```

Invoking parallel execution


```
#pragma acc data copyin(a,b) copyout(c)
{
    ...
    #pragma acc parallel
    {
        #pragma acc loop gang vector
        for (i = 0; i < n; ++i) {
            z[i] = x[i] + y[i];
            ...
        }
        ...
    }
}
```

Providing parameters for loop optimization

Observations

- Single source code needs to be maintained independent of target hardware, or whether sequential or parallel
- Works primarily with loops
- Directives are simple => Small programming overhead
- Performance depends on how the compiler optimises based on directives

Kernels

```
...
```

```
#pragma acc kernels
```

```
{
```

```
    for (i = 0; i < n; ++i) {
```

```
        ...
```

```
    }
```

```
    ...
```

```
    for (j = 0; j < k; ++j) {
```

```
        ...
```

```
    }
```

```
}
```

Everything within this block is to be considered for parallelisation

Kernels

```
...  
  
#pragma acc kernels  
{  
    for (i = 0; i < n; ++i) {  
        ...  
    }  
    ...  
    for (j = 0; j < k; ++j) {  
        ...  
    }  
}
```

Everything within this block is to be considered for parallelisation

Usually this will not perform well. Can you guess why?

```
...
```

```
#pragma acc kernel
```

```
{
```

```
  for (i = 0; i < n; ++i) {
```

```
    ...
```

```
  }
```

```
  ...
```

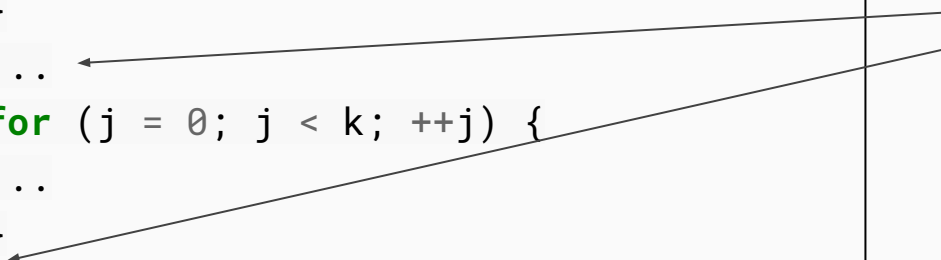
```
    for (j = 0; j < k; ++j) {
```

```
      ...
```

```
    }
```

```
}
```

Conservative assumption is that we have memcopy from device to host at the end of each loop



Kernels

```
...
```

```
#pragma acc data
```

```
{
```

```
    #pragma acc kernels
```

```
    ...
```

```
    #pragma acc kernels
```

```
    ...
```

```
}
```

Declares that the data will remain in device until the end of this block

Different clauses for the data directive

Clause	Interpretation in the CUDA semantics we are familiar with
copy	cudaMalloc -> memcpy h2d at entry -> memcpy d2h at exit
copyin	cudaMalloc -> memcpy h2d at entry
copyout	cudaMalloc -> memcpy d2h at exit
create	cudaMalloc
present	cudaMalloc preserved beyond life time of data block

Data with clauses

...

```
#pragma acc data copyin(A) create(temp)  
copyout(B)
```

```
{
```

```
    #pragma acc kernels
```

```
    for (i = 0; i < n; ++i) {  
        temp[i] = f(A[i])
```

```
    }
```

```
    ...
```

```
    for (j = 0; j < n; ++i) {  
        B[i] = g(temp[i])
```

```
    }
```

```
}
```


The loop directive

```
...  
  
#pragma acc kernel  
{  
    #pragma acc loop  
    for (i = 0; i < n; ++i) {  
        ...  
    }  
    ...  
    #pragma acc loop  
    for (j = 0; j < k; ++j) {  
        ...  
    }  
}
```

Explicitly call out that a loop
can be parallelised and that
each iteration is independent
Helps with pointer aliasing

The parallel directive

```
...  
{  
    #pragma acc parallel loop  
    for (i = 0; i < n; ++i) {  
        ...  
    }  
    ...  
    #pragma acc parallel loop  
    for (j = 0; j < k; ++j) {  
        ...  
    }  
}
```

Programmer tells the compiler that it is definitely possible to parallelise these loops -> Programmer's responsibility

- So far we have had no control on how the parallel execution occurs
- OpenACC provides ways to specify for each parallel block some options on how to parallelise
- Define gang, worker, vector roughly parallel to CUDA blocks, warps, and threads
- `#pragma acc loop gang(32), vector(16)` - use 32 blocks, 16 threads

Other stuff

- Loops support reduction operations
- Supports pipeline with async memcopies
- Support multiple heterogeneous devices
- ...