Code generation and data sharing for OpenMP offloading to NVIDIA devices

12/01/2017

Gheorghe-Teodor Bercea

Postdoctoral Researcher

IBM TJ Watson Research Center

Yorktown Heights, NY

Good practices for device offloading

Managing data

- Data transfers **to** and **from** the device have a high impact on the overall application performance.
- Data should be moved between host and device as few times as possible. In some cases it is enough to map data to the device once.
- A typical use case is represented by time-stepping loops:

```
#pragma omp target enter data map(to:a[:N])
while(time < T){
  #pragma omp target data map(to:a[:N])
  #pragma omp target teams distribute parallel for
  for(int i=0; i<N; i++){
    ...
  }
  #pragma omp target data map(to:a[:N])
  #pragma omp target teams distribute
  for(int i=0; i<N; i++){
    ...
  }
  time += timestep;
}
#pragma omp target exit data map(from:a[:N])
```

Mapping data to devices

- Data accessed on the device is mapped:
  - **implicitly** - any variable whose size in memory can be determined at compile time. Pointers are always mapped implicitly, the data they point to may be mapped implicitly if the size is known at compile time also.
  - **explicitly**:

```
#pragma omp target data map(to: a[0:100])
#pragma omp target data map(from: a[0:100])
#pragma omp target data map(tofrom: a[0:100])

#pragma omp target enter data map(to: a[0:10])
#pragma omp target enter data map(alloc: a[0:10])

#pragma omp target exit data map(from: a[0:10])
#pragma omp target exit data map(delete: a[0:10])

#pragma omp target update to(a[0:10])
#pragma omp target update from(a[0:10])

#pragma omp declare target
#pragma omp end declare target
```

Inlining of runtime functions

- Single parallel loops are amenable to the generation of more efficient code. They do not require any calls to the OpenMP runtime and the use of any shared data.
- In the latest version of the CLANG/LLVM compiler, code generation has been modified to reuse as much of the existing host code generation infrastructure as possible. Code generation uses runtime calls even for single loops to avoid special casing in the compiler.
- To keep overheads low, the runtime functions are inlined and any redundant code is optimized out by the LLVM backend passes.
- The code that is obtained at the end of the compilation process for single loops matches the code previously obtained in an earlier version of the compiler where these type of loops were special cased.
- **Inlining is ensured by having the `libomptarget-nvptx.bc` library in the `${CLANG_OMP_LIB}` folder.**
  - Build CLANG with the following additional cmake options:

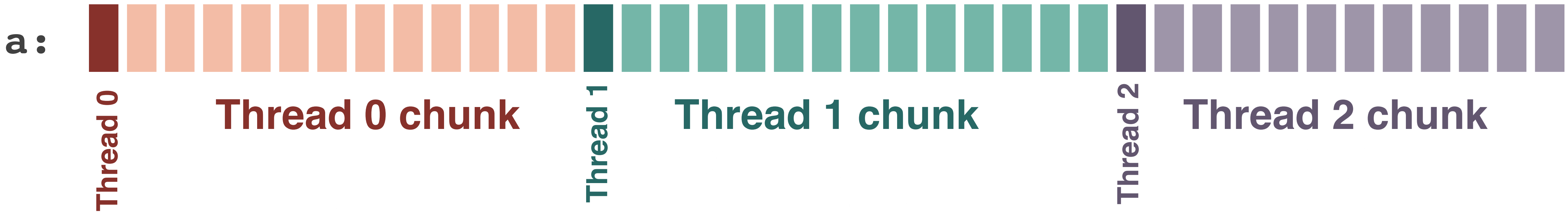
```
87-DLIBOMPTARGET_NVPTX_ENABLE_BCLIB=true \

-DLIBOMPTARGET_NVPTX_CUDA_COMPILER=${LLVM_OBJ}/bin/clang \

-DLIBOMPTARGET_NVPTX_BC_LINKER=${LLVM_OBJ}/bin/llvm-link \
```

Single parallel loops - host

```
int N = 100;
double *a = (double*)
  malloc (sizeof(double)*N);
#pragma omp parallel for
for(int i=0; i<N; i++){
     a[i] = i;
}
```

- This loop will be executed on the host.
- Iterations will be assigned to threads as uniformly as possible to exploit locality.
- Example: if the number of threads is equal to 3 then:
  - iterations 0-33 are assigned to thread 0
  - iterations 34-66 are assigned to thread 1
  - iterations 67-99 are assigned to thread 2

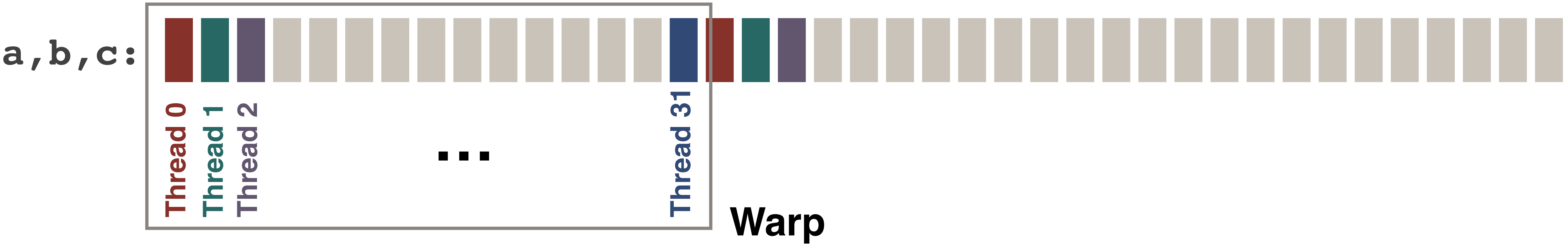
The default chunk size on host is: ***N / num_threads*** *(great for the host side).*



Single parallel loops - device

```
int N = 100;
double *a = (double*) malloc (sizeof(double) * N);
double *b = (double*) malloc (sizeof(double) * N);
double *c = (double*) malloc (sizeof(double) * N);
#pragma omp target data map(to:a[:N], b[:N]) map(from:c[:N])
#pragma omp target teams distribute parallel for
for(int i=0; i<N; i++){
     c[i] = a[i] + alfa*b[i];
}
```

- Single data parallel loops are the most suitable OpenMP loops for GPU execution.
- Data accesses on NVIDIA GPUs are typically grouped together (**coalesced**) into a single data access to make full use of the available bandwidth. Data accessed by threads within a warp can be *coalesced* in a single memory transaction if **all warp threads access consecutive memory locations.**
- The **default schedule on the device** (equivalent to the `schedule(static, 1)` clause) assigns a loop iteration to each thread in a round-robin fashion hence the accesses to arrays **a**, **b** and **c**:



Use collapse(k) clause

The **`for`** directive has a clause called **`collapse(k)`.**

A perfect loop nest can be collapsed using the **`collapse(k)`** clause where **`k`** represents the number of collapsed loops on both **host** and **device**.

```
#pragma omp parallel for collapse(3)
for(int i=0; i<N; i++){
     for(int j=0; j<M; j++){
          for(int k=0; k<M; k++){
          }
     }
}
```

Nested parallel loops

- There are situations in which achieving performance relies on being able to parallelize over more than one loop.
- Such application can be found among the CORAL proxy apps such as HACCMK.
- More common examples are provided by parallel loops with inner SIMD-izable loops.

```
#pragma omp parallel for collapse(3)
for(int i=0; i<N; i++){
  for(int j=0; j<M; j++){
    for(int k=0; k<P; k++){
      double a = omp_get_thread_num();
      #pragma omp simd
      for(int l=0; l<1000; l++){
        b[l] = sin(a) * c[l];
      }
    }
  }
}
```

Nested parallelism

Code generation

- The current CLANG/LLVM implementation makes use of a code generation scheme which relies on **function outlining**.
- The master thread (actually the whole master warp) executes separately from the worker threads. Communication between master and worker threads occurs via shared variables.

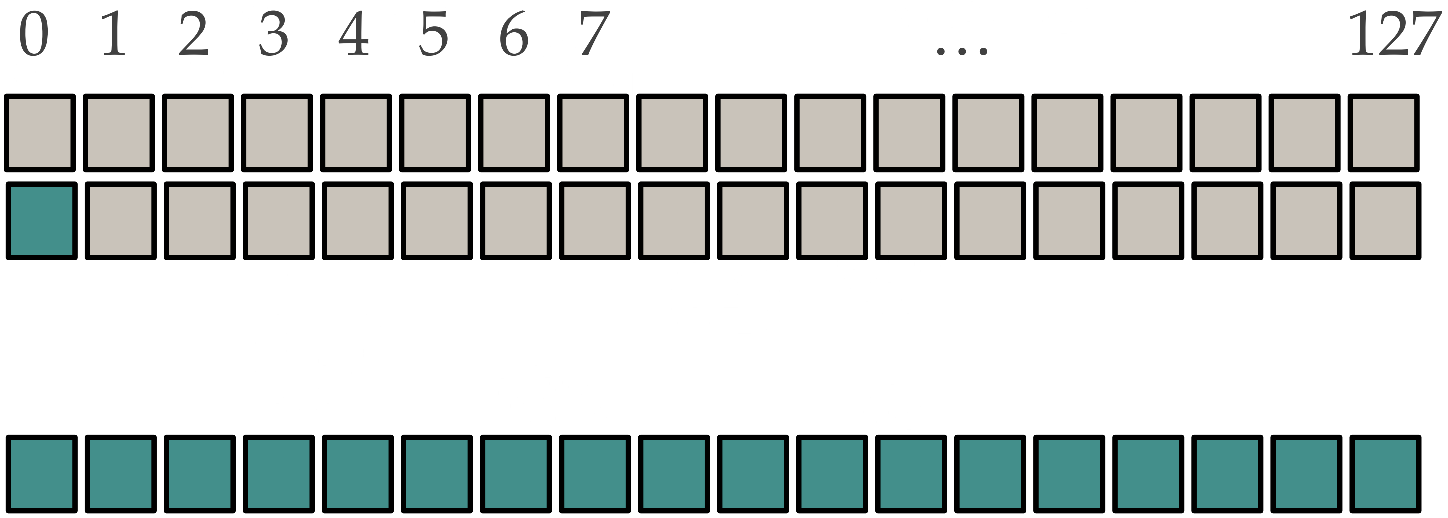
Data sharing

- OpenMP often requires regions of code to be executed by a **single thread**. Due to side effects, this code might not be safe to execute in parallel (redundantly).
- Data sharing occurs:
  - data shared among teams - global data
  - data shared between threads in a team
  - data shared among the threads within a warp

Level0 data sharing

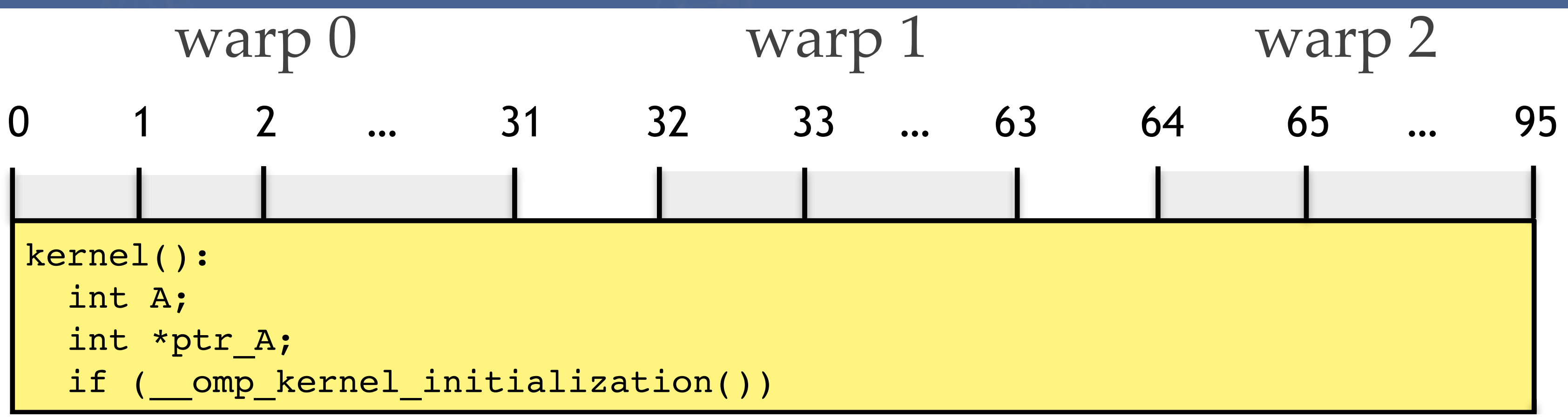
Instances of **A** for kernel launched with **1 team** and **128 threads per team**

```
#pragma omp target
{
  int A;
  // update A
  #pragma omp parallel for
  for(…) {
    // use A
  }
}
```



Level0 - team master to threads

```
#pragma omp target
{
  int A
  // S1 using A
  #pragma omp parallel
  {
     // S2 using A
  }
  // S3 using A
}
```

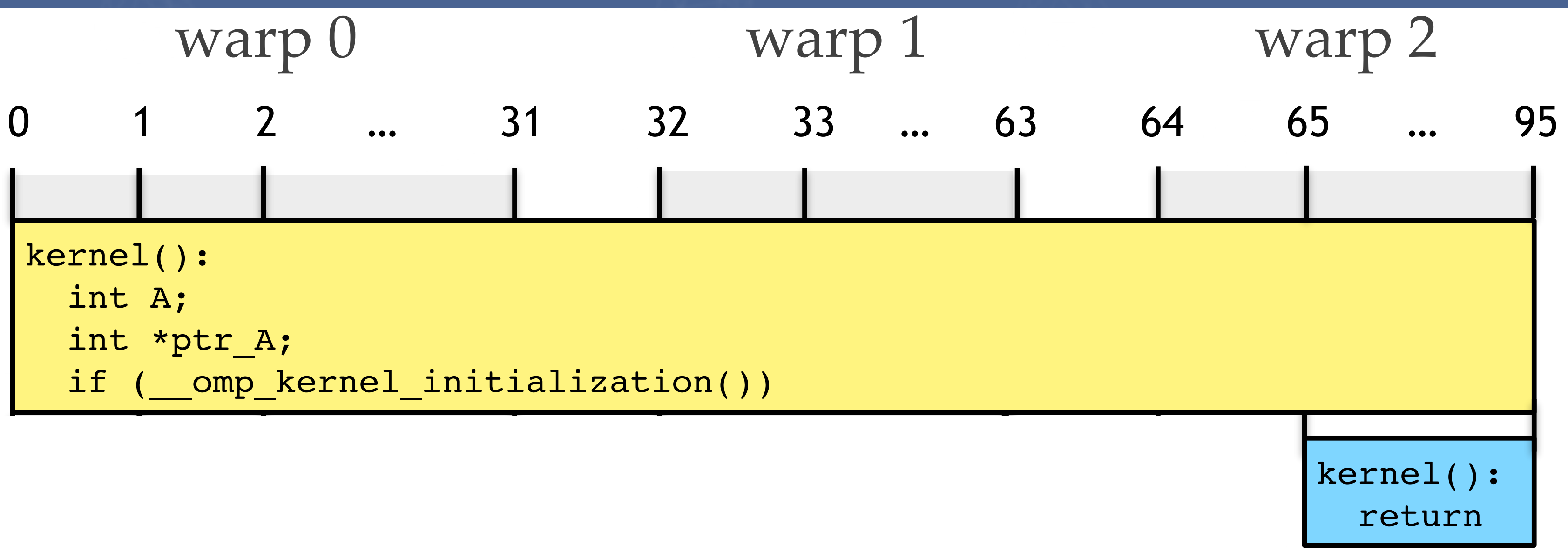


```
kernel():
  int A;
  int *ptr_A;
  if (__omp_kernel_initialization())
```

All the threads start executing the kernel.

Level0 - team master to threads

```
#pragma omp target
{
  int A
  // S1 using A
  #pragma omp parallel
  {
     // S2 using A
  }
  // S3 using A
}
```



```
kernel():
  int A;
  int *ptr_A;
  if (__omp_kernel_initialization())
```

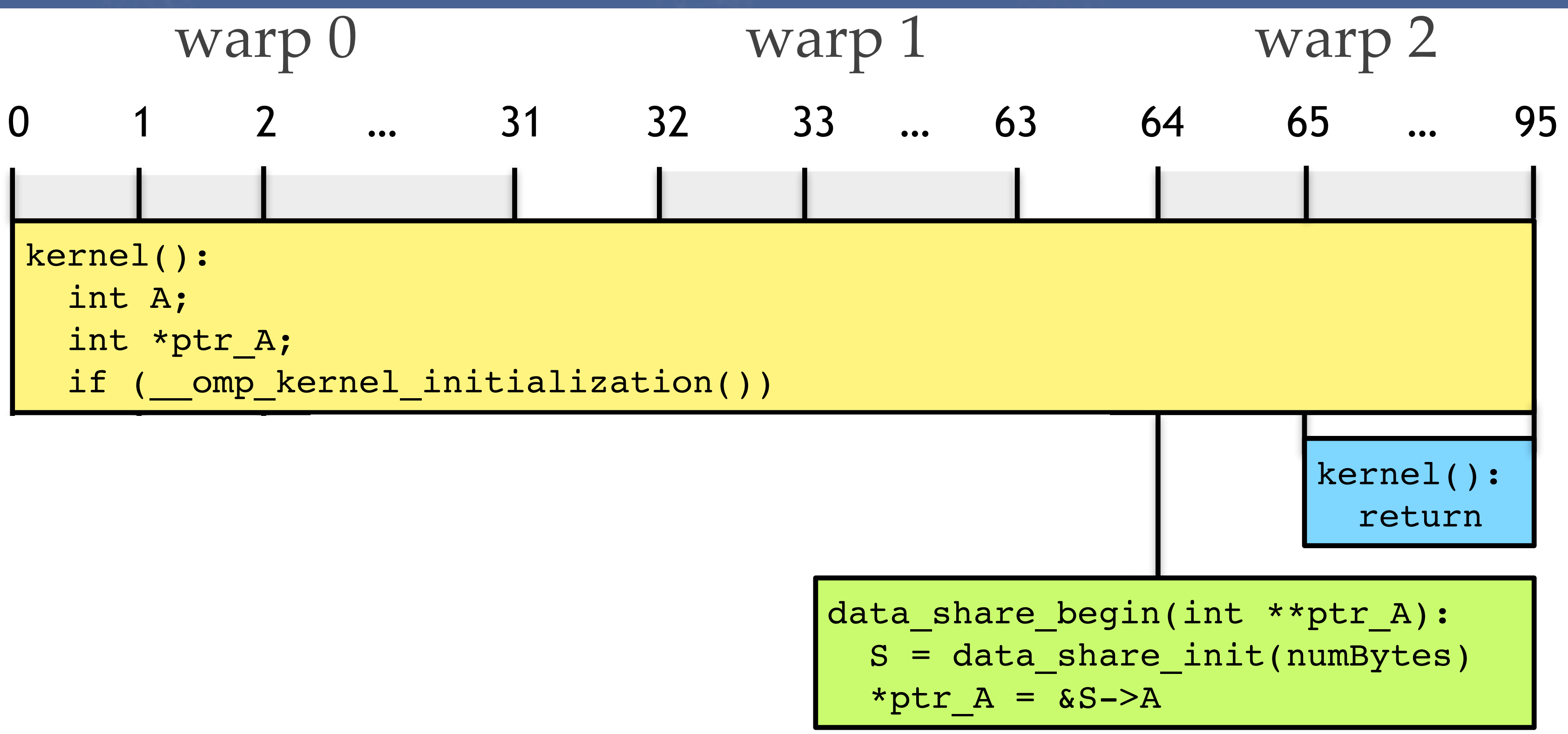
```
kernel():
  return
```

A warp is reserved for the master-only region.

31 of the threads in the master warp return.

Level0 - team master to threads

```
#pragma omp target
{
  int A
  // S1 using A
  #pragma omp parallel
  {
     // S2 using A
  }
  // S3 using A
}
```



```
kernel():
  int A;
  int *ptr_A;
  if (__omp_kernel_initialization())
```

```
kernel():
  return
```

```
data_share_begin(int **ptr_A):
  S = data_share_init(numBytes)
  *ptr_A = &S->A
```

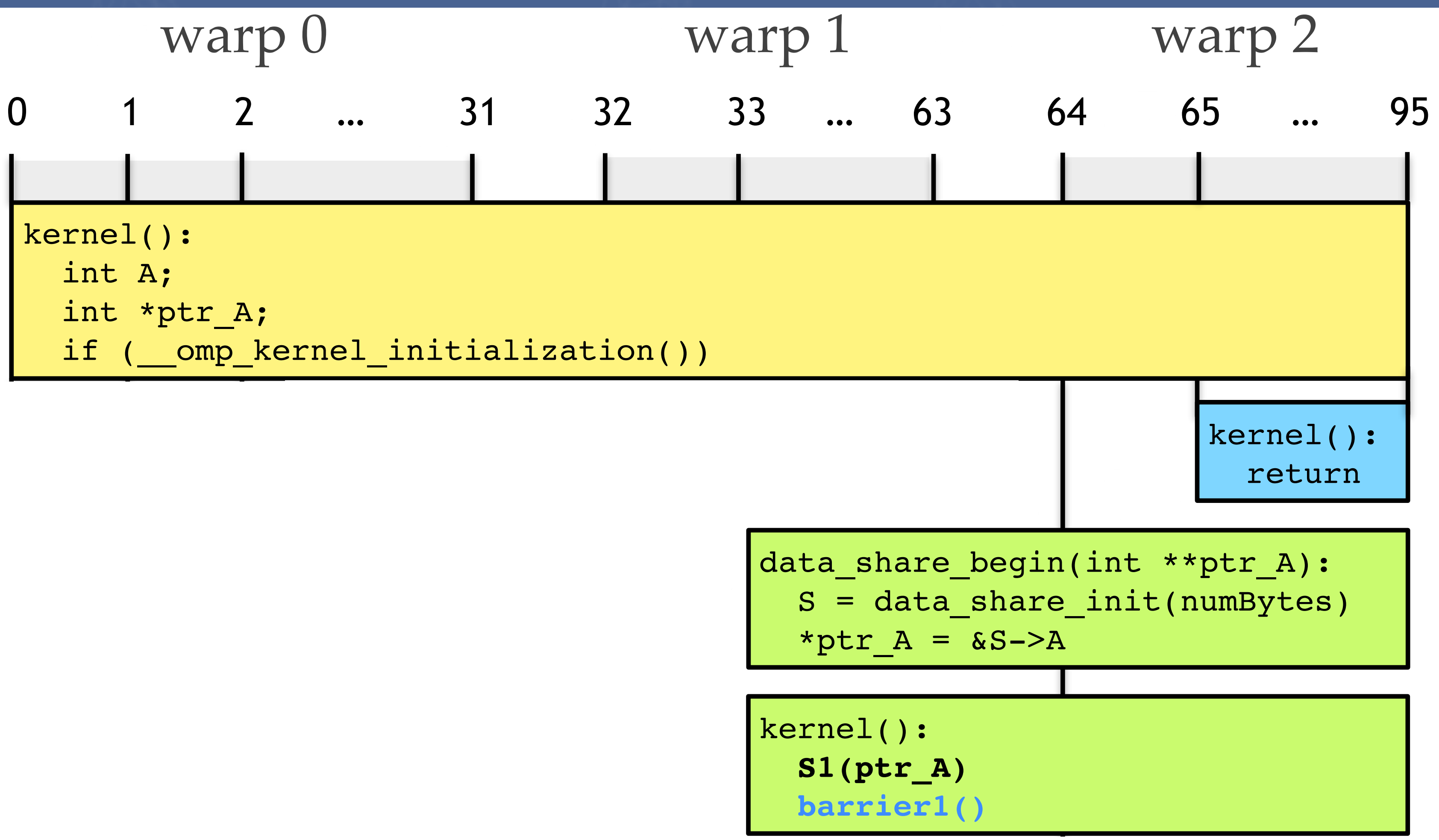
The master thread of the master warp executes the sequential region.

A data sharing region (stack) is initialized.

A pointer to the shared variable A is created.

Level0 - team master to threads

```
#pragma omp target
{
  int A
  // S1 using A
  #pragma omp parallel
  {
     // S2 using A
  }
  // S3 using A
}
```



```
kernel():
  int A;
  int *ptr_A;
  if (__omp_kernel_initialization())
```

```
kernel():
  return
```

```
data_share_begin(int **ptr_A):
  S = data_share_init(numBytes)
  *ptr_A = &S->A
```

```
kernel():
  S1(ptr_A)
  barrier1()
```

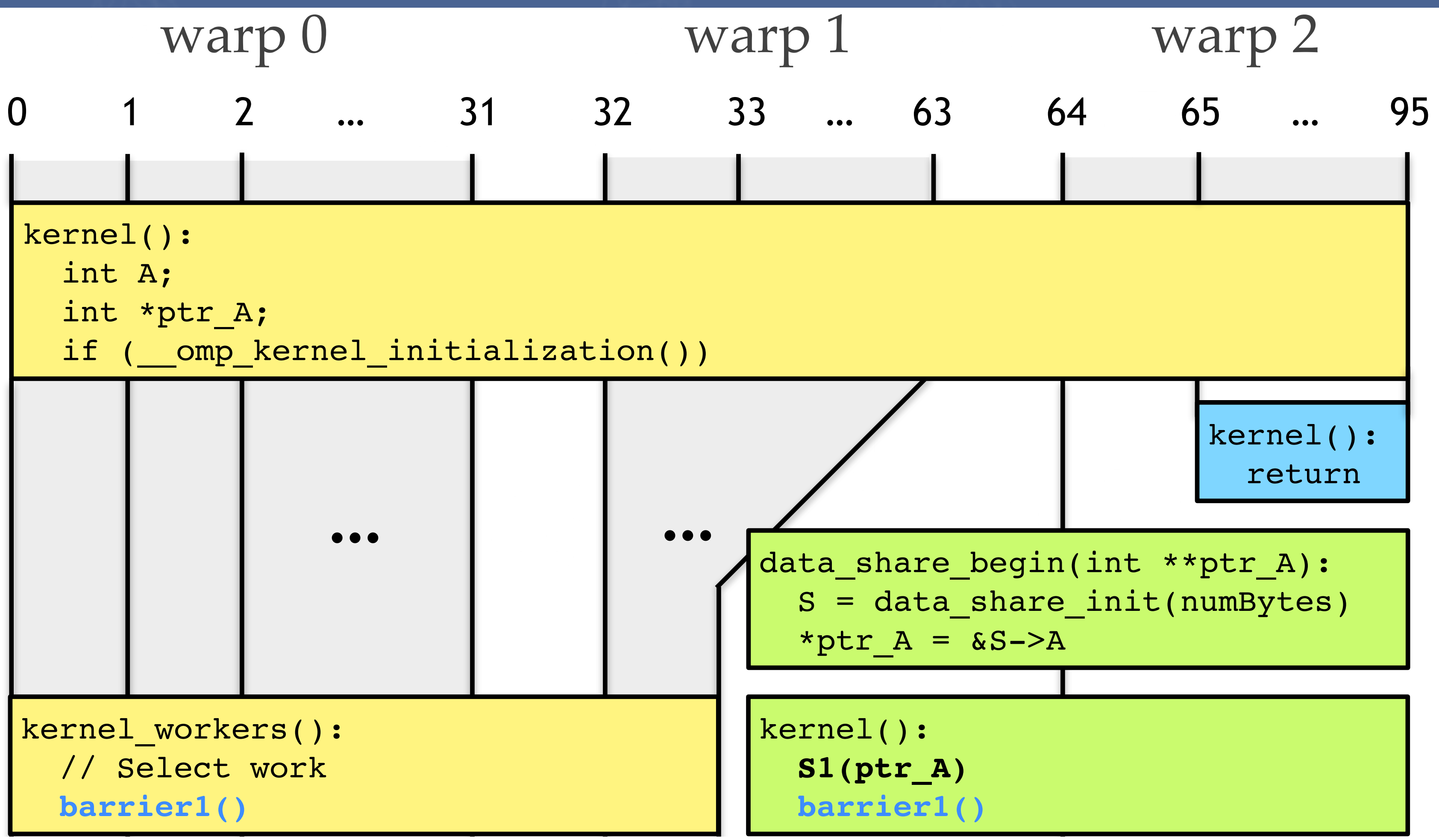
Execution of S1 by a single thread.

Every team executes this region using a single thread.

Any reference in S1 to **A** is replaced with **ptr_A**.

Level0 - team master to threads

```
#pragma omp target
{
  int A
  // S1 using A
  #pragma omp parallel
  {
     // S2 using A
  }
  // S3 using A
}
```



```
kernel():
  int A;
  int *ptr_A;
  if (__omp_kernel_initialization())
```

```
kernel():
  return
```

```
data_share_begin(int **ptr_A):
  S = data_share_init(numBytes)
  *ptr_A = &S->A
```

```
kernel_workers():
  // Select work
  barrier1()
```

```
kernel():
  S1(ptr_A)
  barrier1()
```

The worker threads execute an outlined worker function.

Execution starts once the barrier is reached by the master thread.

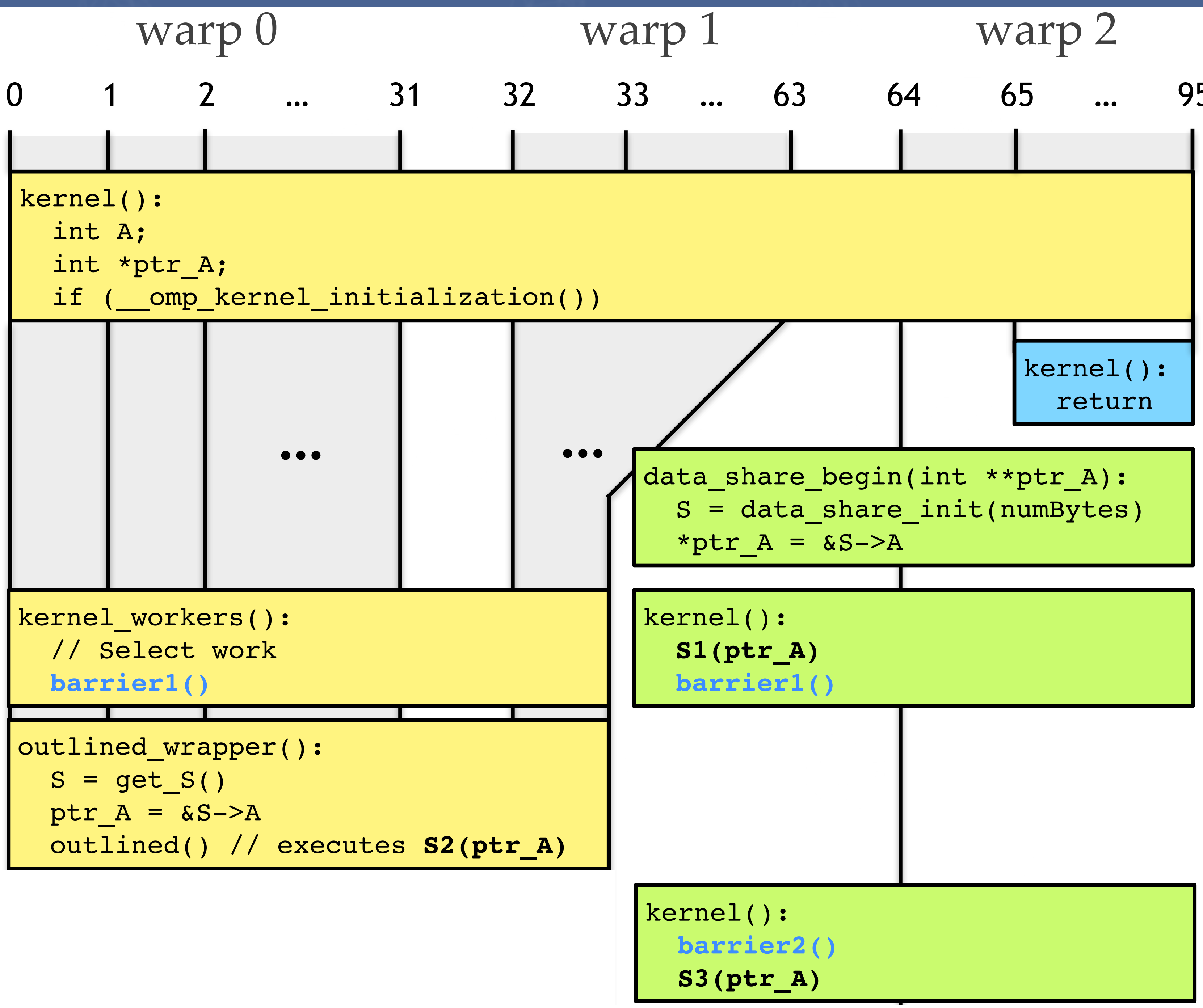
Level0 - team master to threads

```
#pragma omp target
{
  int A
  // S1 using A
  #pragma omp parallel
  {
     // S2 using A
  }
  // S3 using A
}
```

The master thread waits at barrier2.

The worker threads execute the outline worker function.

The worker function retrieves a pointer to shared variable **A** from the shared stack. S2 executes using **ptr_A**.



```
kernel():
  int A;
  int *ptr_A;
  if (__omp_kernel_initialization())
```

```
kernel():
  return
```

```
data_share_begin(int **ptr_A):
  S = data_share_init(numBytes)
  *ptr_A = &S->A
```

```
kernel_workers():
  // Select work
  barrier1()
```

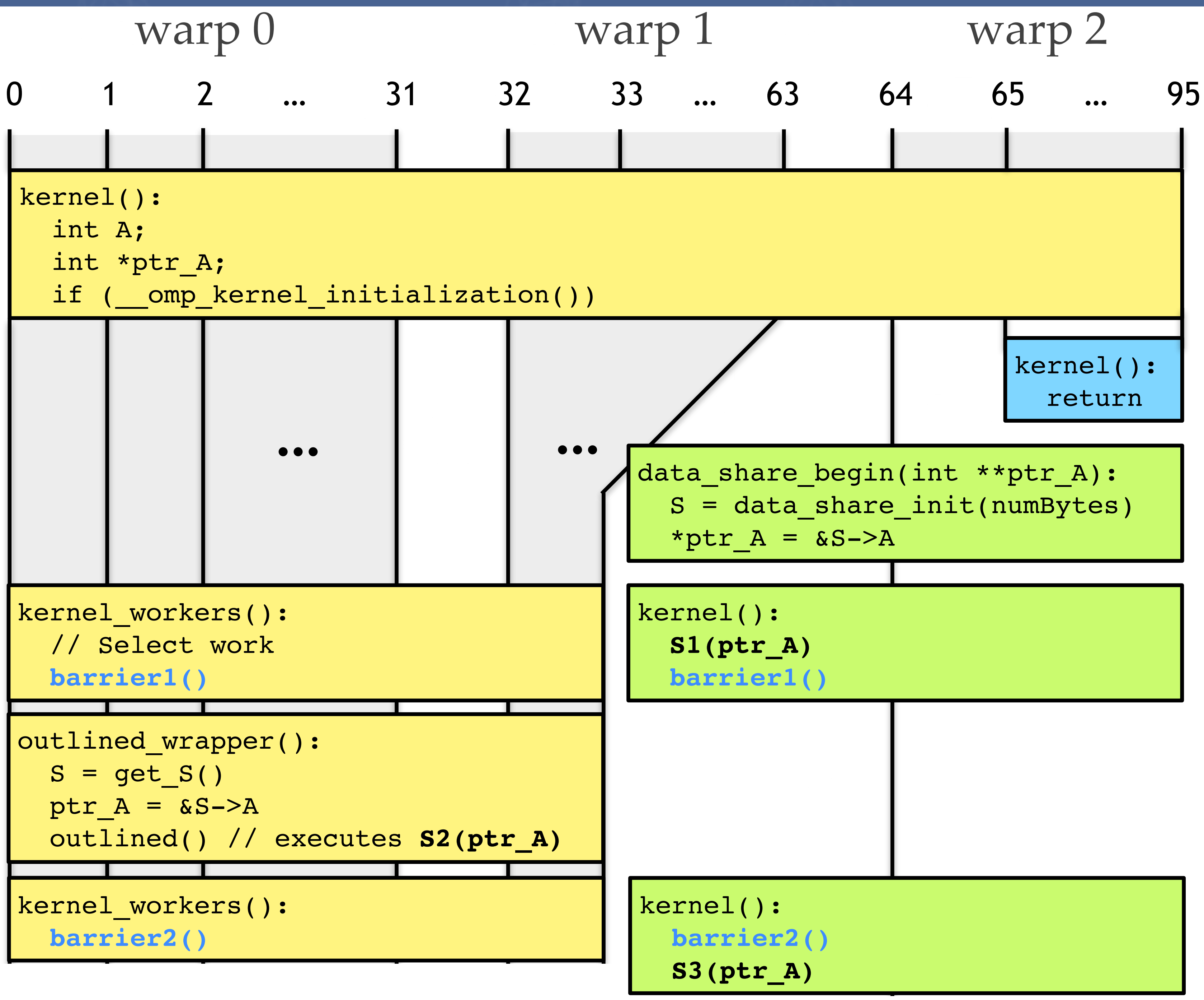
```
kernel():
  S1(ptr_A)
  barrier1()
```

```
outlined_wrapper():
  S = get_S()
  ptr_A = &S->A
  outlined() // executes S2(ptr_A)
```

```
kernel():
  barrier2()
  S3(ptr_A)
```

Level0 - team master to threads

```
#pragma omp target
{
   int A
   // S1 using A
   #pragma omp parallel
   {
      // S2 using A
   }
   // S3 using A
}
```



```
kernel():
  int A;
  int *ptr_A;
  if (__omp_kernel_initialization())
```

```
kernel():
  return
```

```
data_share_begin(int **ptr_A):
  S = data_share_init(numBytes)
  *ptr_A = &S->A
```

```
kernel_workers():
  // Select work
  barrier1()
```

```
kernel():
  S1(ptr_A)
  barrier1()
```

```
outlined_wrapper():
  S = get_S()
  ptr_A = &S->A
  outlined() // executes S2(ptr_A)
```

```
kernel_workers():
  barrier2()
```

```
kernel():
  barrier2()
  S3(ptr_A)
```

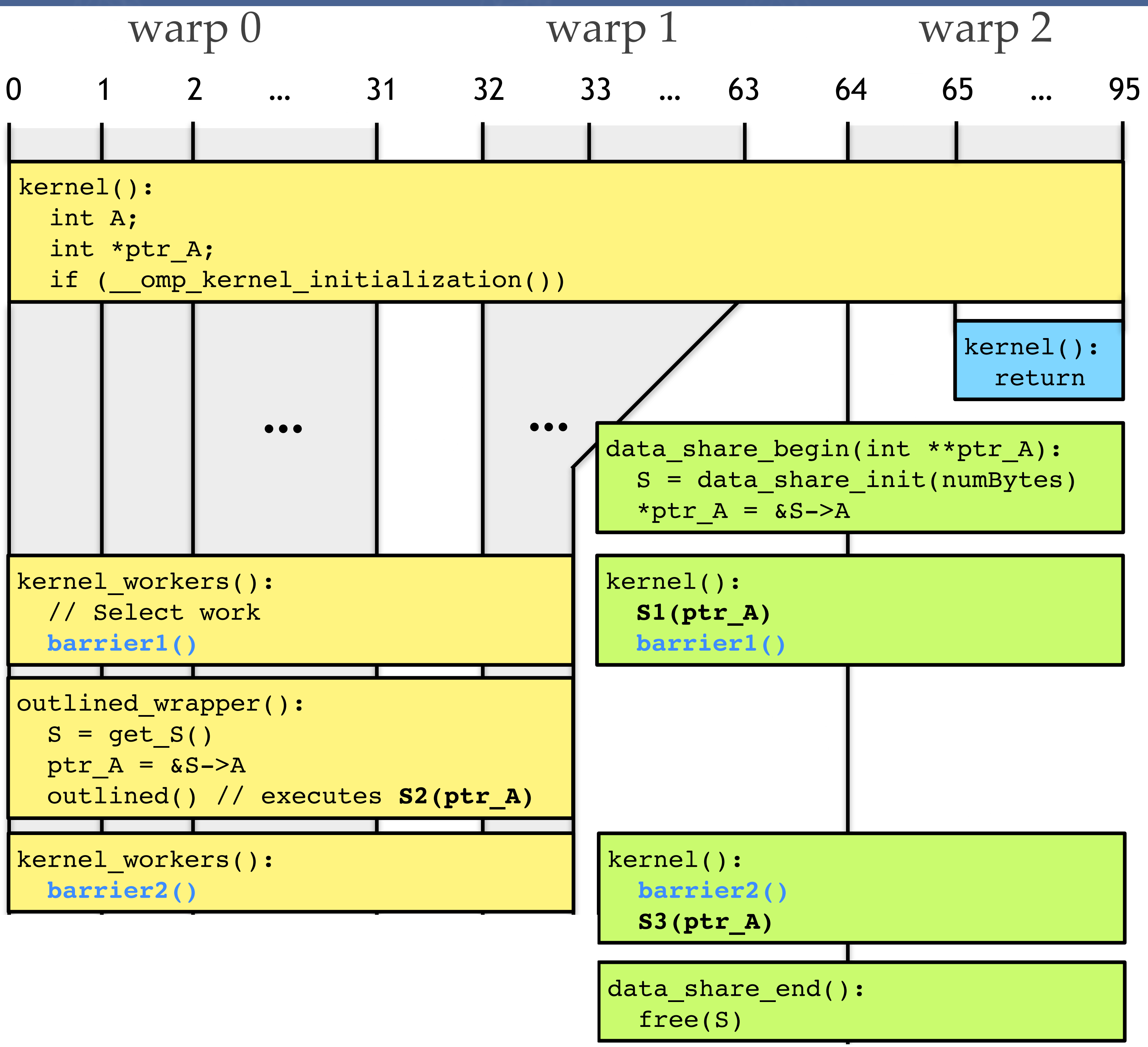
The workers hit barrier2 and unlock the master thread.

The master thread executes S3 using **ptr_A**.

Level0 - team master to threads

```
#pragma omp target
{
  int A
  // S1 using A
  #pragma omp parallel
  {
    // S2 using A
  }
  // S3 using A
}
```

Clean-up any data sharing.



```
kernel():
  int A;
  int *ptr_A;
  if (__omp_kernel_initialization())
```

```
kernel():
  return
```

```
data_share_begin(int **ptr_A):
  S = data_share_init(numBytes)
  *ptr_A = &S->A
```

```
kernel_workers():
  // Select work
  barrier1()
```

```
kernel():
  S1(ptr_A)
  barrier1()
```

```
outlined_wrapper():
  S = get_S()
  ptr_A = &S->A
  outlined() // executes S2(ptr_A)
```

```
kernel_workers():
  barrier2()
```

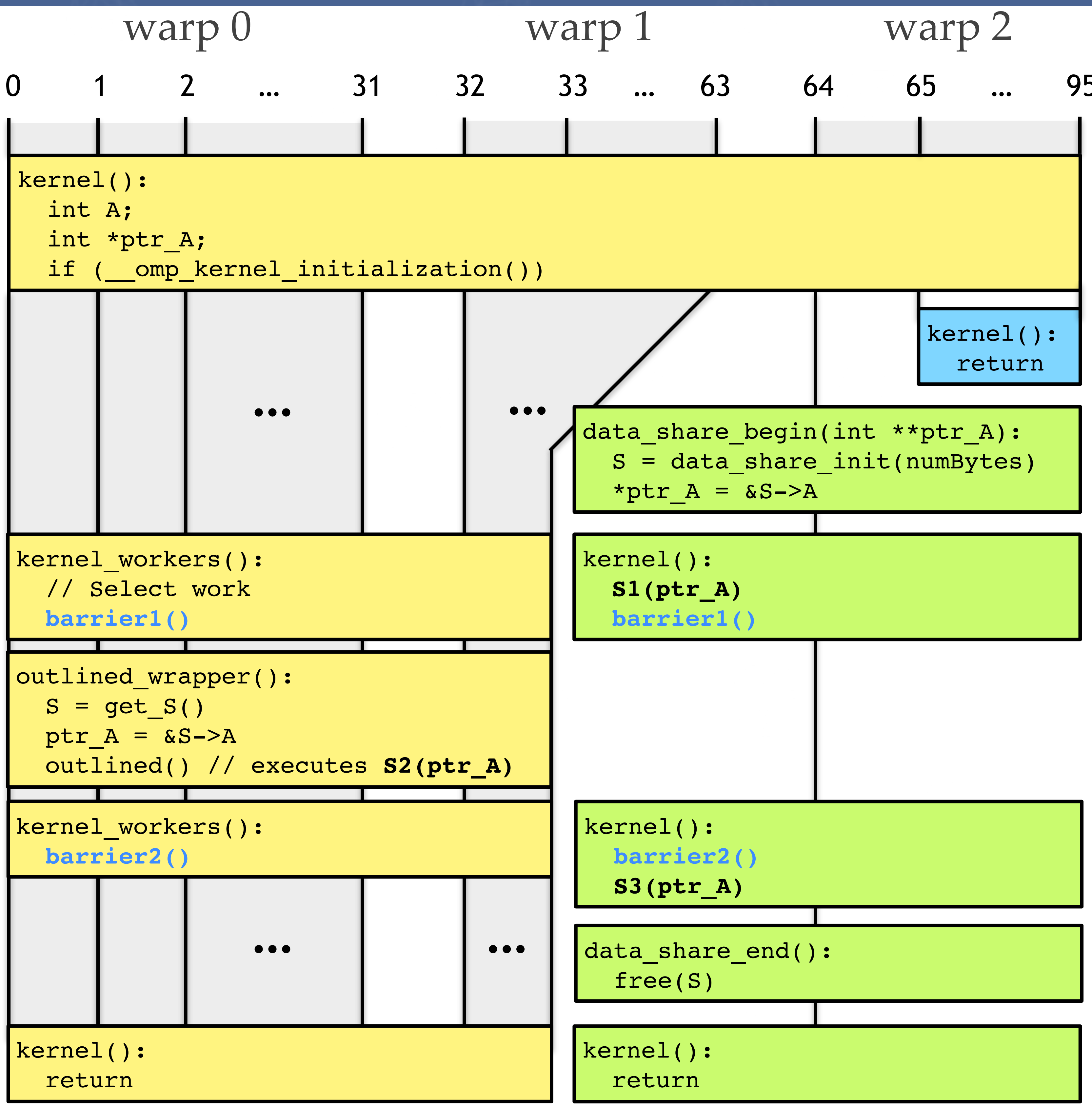
```
kernel():
  barrier2()
  S3(ptr_A)
```

```
data_share_end():
  free(S)
```

Level0 - team master to threads

```
#pragma omp target
{
  int A
  // S1 using A
  #pragma omp parallel
  {
     // S2 using A
  }
  // S3 using A
}
```

All threads return.



```
kernel():
  int A;
  int *ptr_A;
  if (__omp_kernel_initialization())
```

```
kernel():
  return
```

```
data_share_begin(int **ptr_A):
  S = data_share_init(numBytes)
  *ptr_A = &S->A
```

```
kernel_workers():
  // Select work
  barrier1()
```

```
kernel():
  S1(ptr_A)
  barrier1()
```

```
outlined_wrapper():
  S = get_S()
  ptr_A = &S->A
  outlined() // executes S2(ptr_A)
```

```
kernel_workers():
  barrier2()
```

```
kernel():
  barrier2()
  S3(ptr_A)
```

```
data_share_end():
  free(S)
```

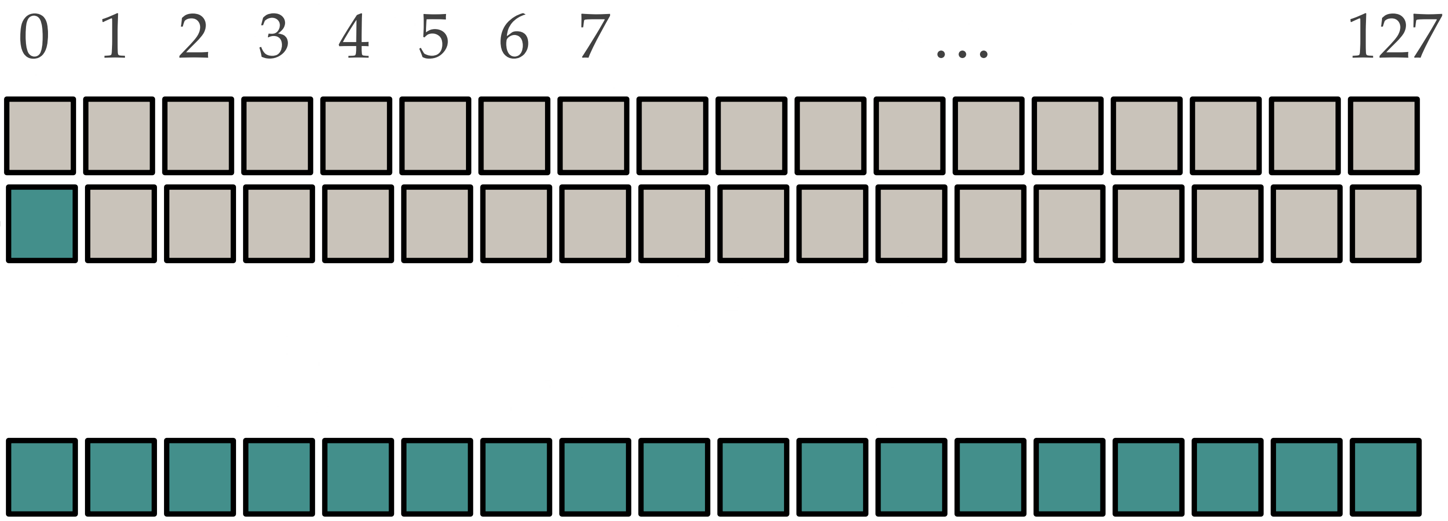
```
kernel():
  return
```

```
kernel():
  return
```

Level0 data sharing

Instances of **A** for kernel launched with **1 team** and **128 threads per team**

```
#pragma omp target
{
  int A;
  // update A
  #pragma omp parallel for
  for(…) {
    // use A
  }
}
```



There is one shared value of **A** updated by the team master.
The rest of the threads in the team have a pointer to the shared value of **A**.

Level1 data sharing

- Occurs between threads within a warp.
- This type of sharing occurs in **OpenMP nested parallel regions**.
- Region **S2 must be executed by all threads** in order to achieve performance.
- Region **S1 may benefit from being executed by all threads**.
- Regions executed by all threads must ensure **coalesced memory accesses**.
- To avoid synchronization overhead incurred by barriers, **the sharing is contained within each warp** executing the nested parallel region. Memory fences - which incur less overhead - are enough to ensure correct data updates.

```
#pragma omp target
{
  int A;
  #pragma omp parallel
  {
     int B;
     // S1 using A, B
     #pragma omp parallel
     {
        // S2 using B
     }
  }
}
```

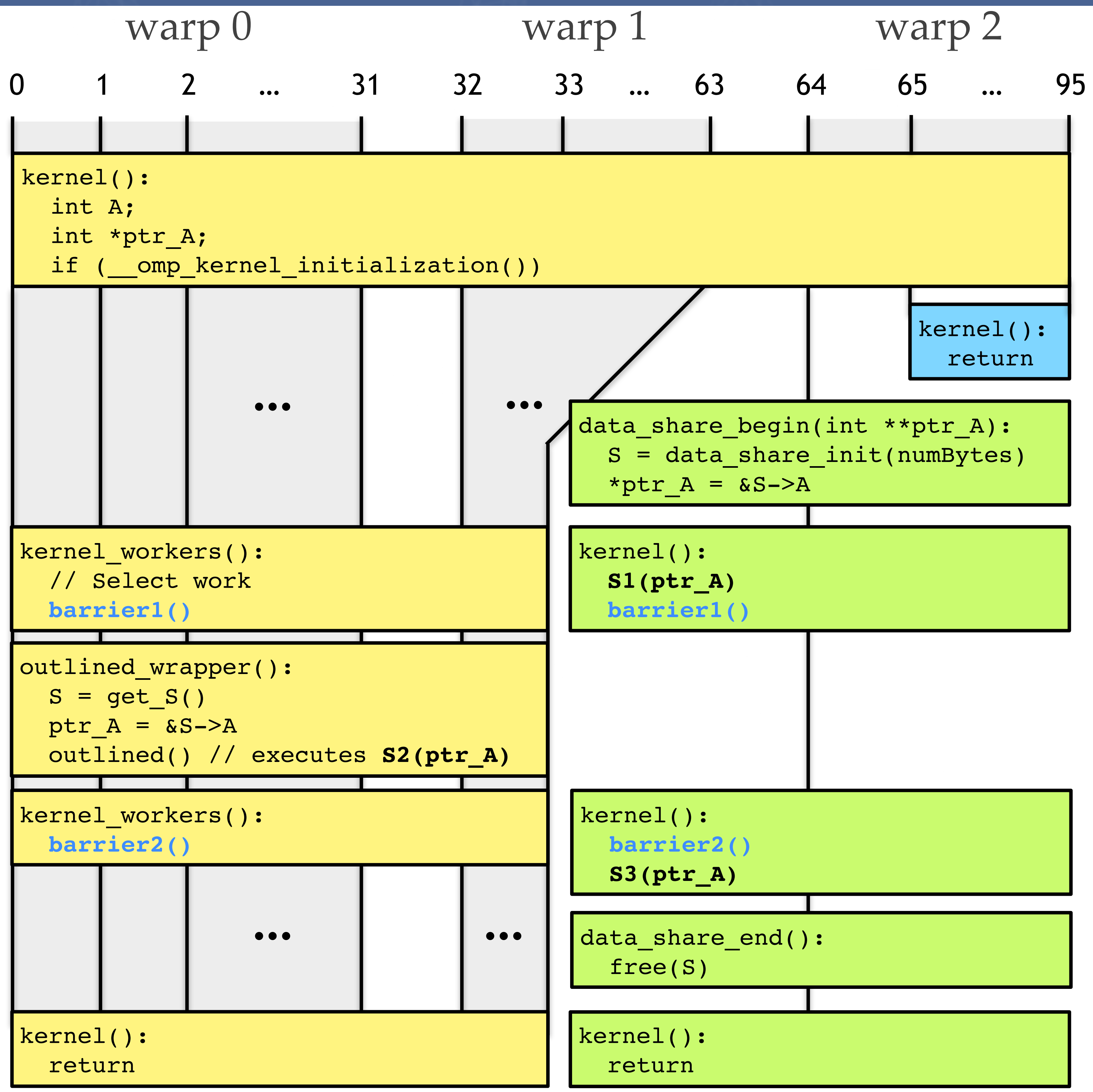
Level1 data sharing

- All 32 threads in a warp execute region S1 and hit the inner parallel construct.
- There are several ways in which the execution can happen next:
  - each thread executes the parallel region sequentially - no way of achieving coalescing, each thread executes all the innermost iterations.
  - all the threads in the warp execute the inner parallel encountered by thread 0 in the warp. Then all threads in the warp execute the inner parallel hit by thread 1 in the warp and so on. Coalescing is now guaranteed.

```
#pragma omp target
{
  int A;
  #pragma omp parallel
  {
     int B;
     // S1 using A, B
     #pragma omp parallel
     {
        // S2 using B
     }
  }
}
```

Level0 - team master to threads

```
#pragma omp target
{
  int A
  #pragma omp parallel
  {
     // using A
     ...
  }
}
```



```
kernel():
  int A;
  int *ptr_A;
  if (__omp_kernel_initialization())
```

```
kernel():
  return
```

```
data_share_begin(int **ptr_A):
  S = data_share_init(numBytes)
  *ptr_A = &S->A
```

```
kernel_workers():
  // Select work
  barrier1()
```

```
kernel():
  S1(ptr_A)
  barrier1()
```

```
outlined_wrapper():
  S = get_S()
  ptr_A = &S->A
  outlined() // executes S2(ptr_A)
```

```
kernel_workers():
  barrier2()
```

```
kernel():
  barrier2()
  S3(ptr_A)
```

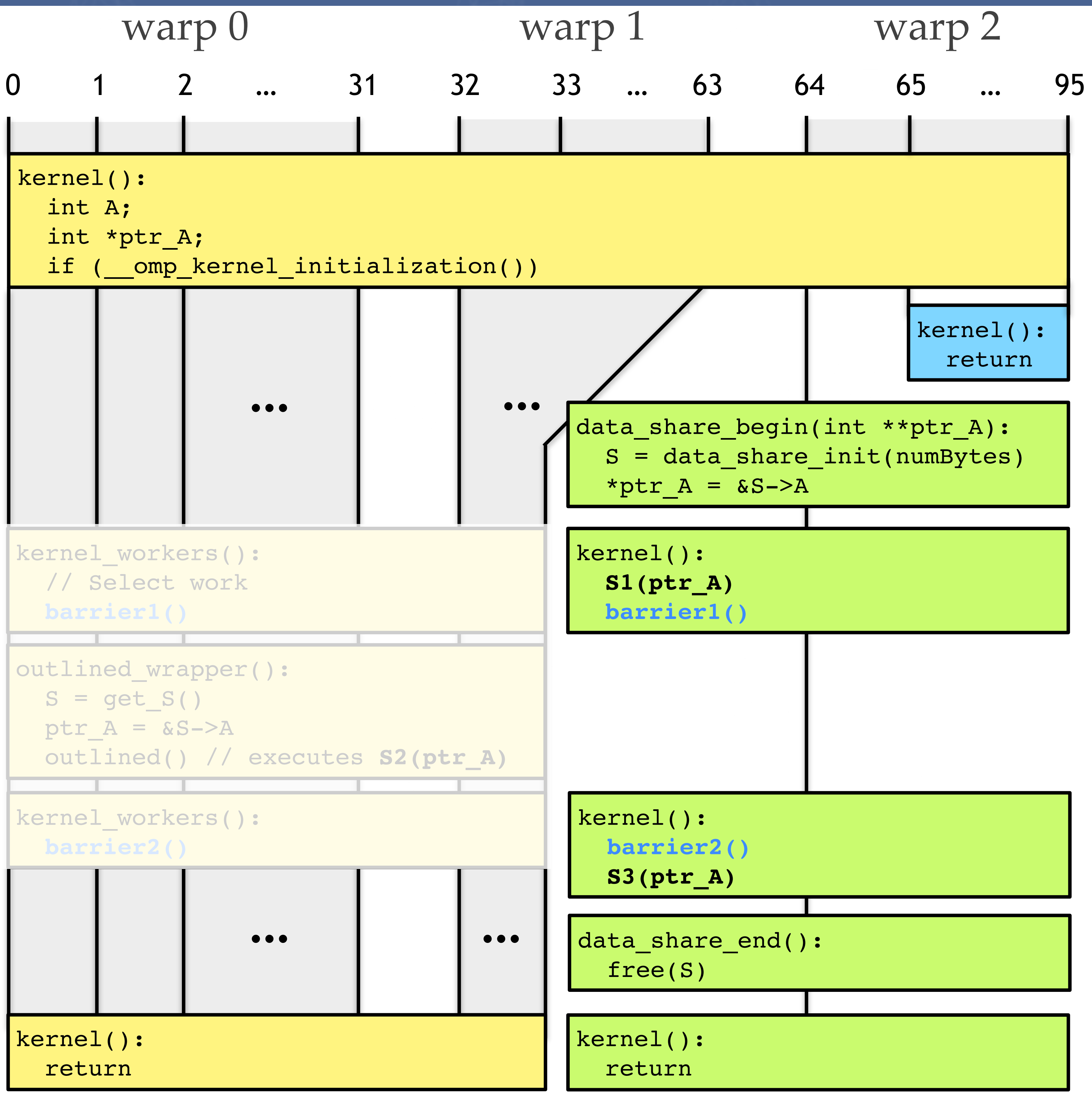
```
data_share_end():
  free(S)
```

```
kernel():
  return
```

```
kernel():
  return
```

Level1 - warp threads

```
#pragma omp target
{
  int A;
  #pragma omp parallel
  {
     int B;
     // S1 using A, B
     #pragma omp parallel
     {
        // S2 using B
     }
  }
}
```



```
kernel():
  int A;
  int *ptr_A;
  if (__omp_kernel_initialization())
```

```
kernel():
  return
```

```
data_share_begin(int **ptr_A):
  S = data_share_init(numBytes)
  *ptr_A = &S->A
```

```
kernel_workers():
  // Select work
  barrier1()
```

```
kernel():
  S1(ptr_A)
  barrier1()
```

```
outlined_wrapper():
  S = get_S()
  ptr_A = &S->A
  outlined() // executes S2(ptr_A)
```

```
kernel_workers():
  barrier2()
```

```
kernel():
  barrier2()
  S3(ptr_A)
```

```
data_share_end():
  free(S)
```

```
kernel():
  return
```

```
kernel():
  return
```

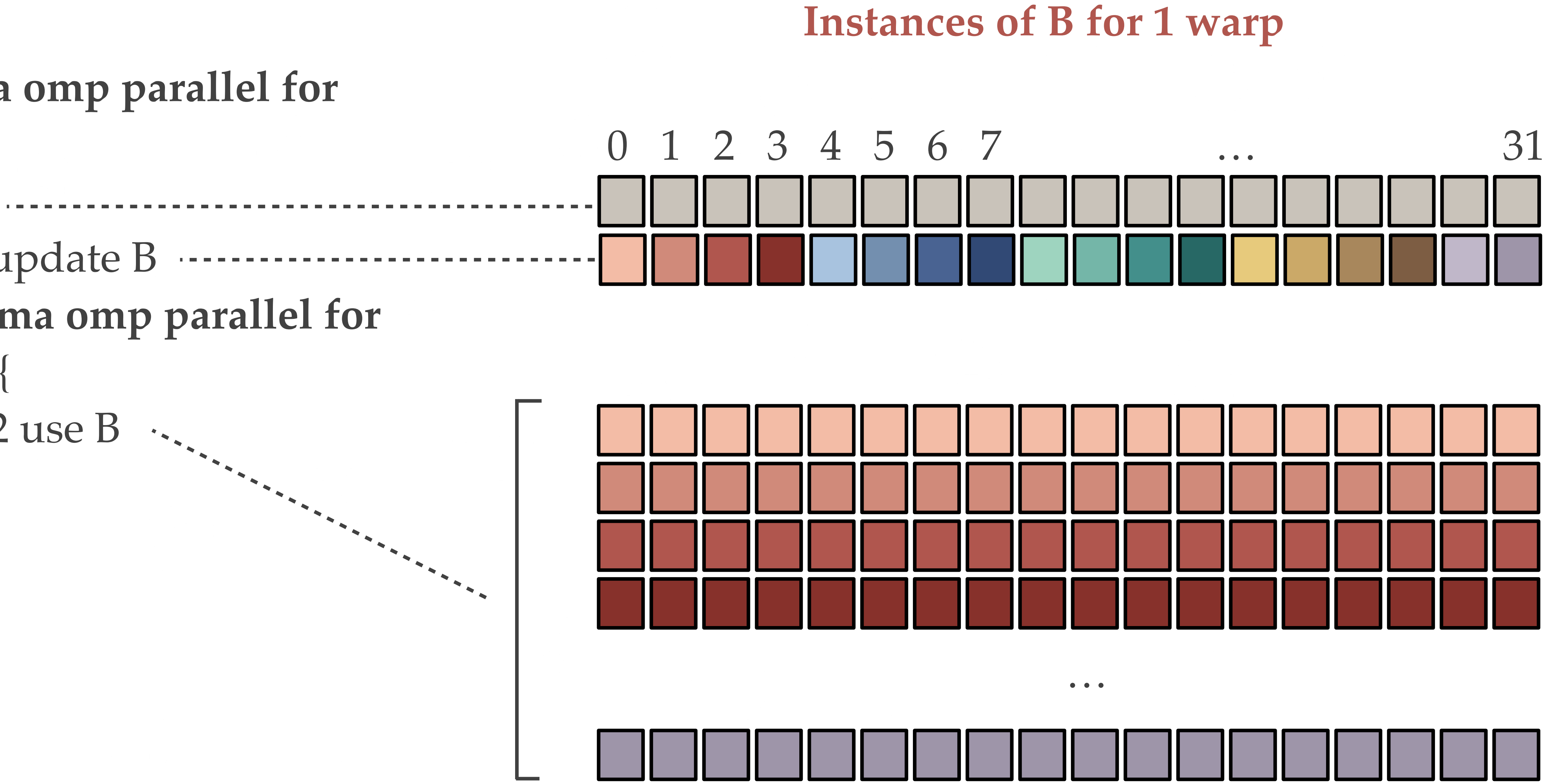
Sharing of A is the same.

Only focus on 1 warp.

Level1 data sharing

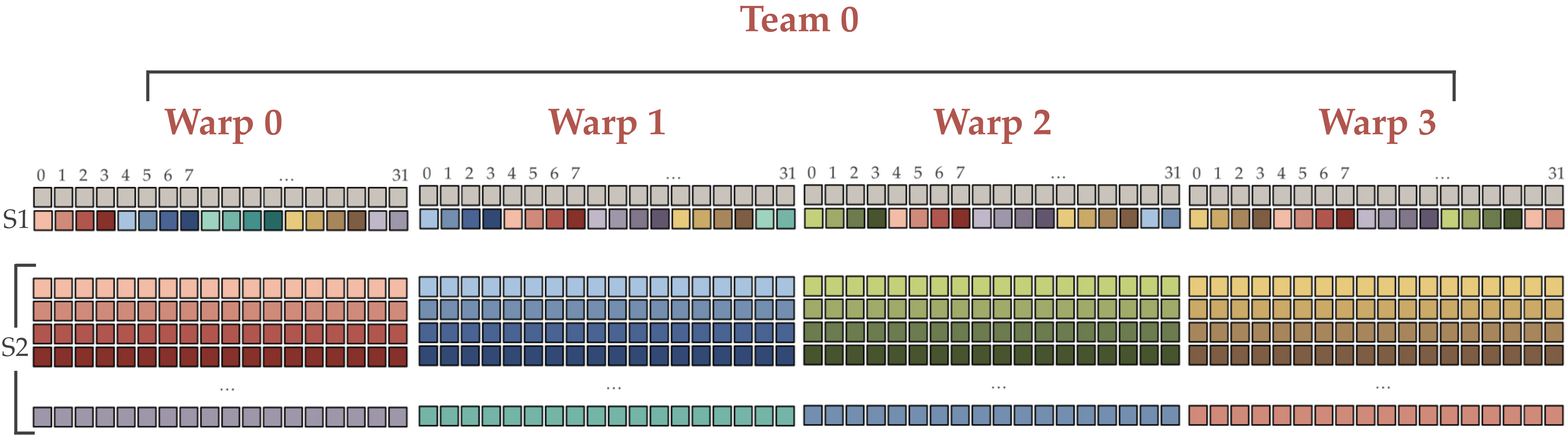
Instances of **B** for kernel launched with **1 team** and **128 threads per team**

```
#pragma omp target
{
  #pragma omp parallel for
  for(…) {
    int B;
    // S1 update B
    # pragma omp parallel for
    for(…){
      // S2 use B
    }
  }
}
```



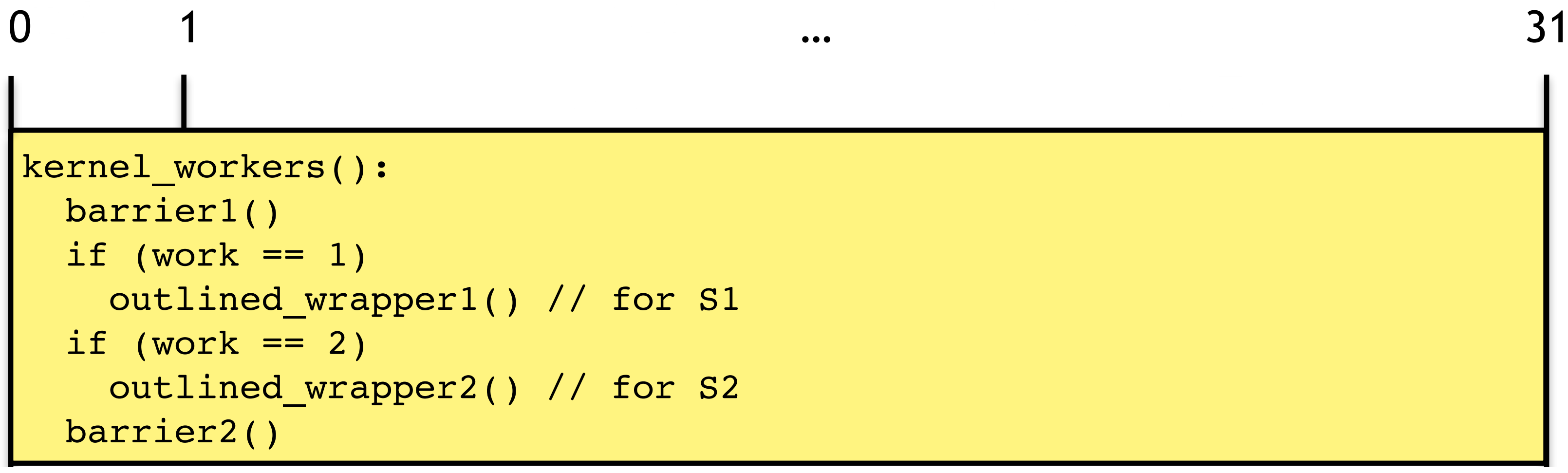
Level1 data sharing - all warps

Assuming that there are more than 128 iterations in the loop all 4 warps will run in parallel



Level1 - warp threads

```
#pragma omp target
{
  int A;
  #pragma omp parallel
  {
     int B;
     // S1 using A, B
     #pragma omp parallel
     {
        // S2 using B
     }
  }
}
```

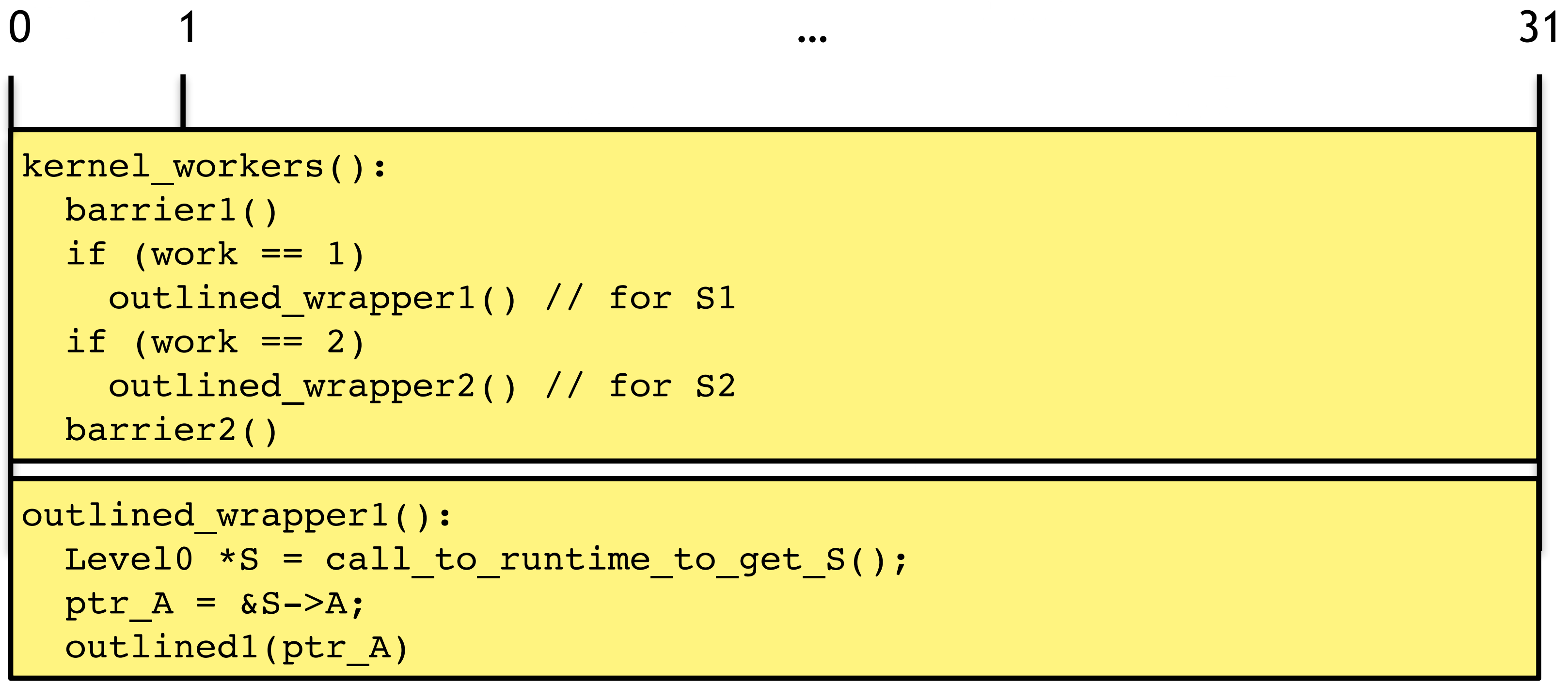


```
kernel_workers():
  barrier1()
  if (work == 1)
    outlined_wrapper1() // for S1
  if (work == 2)
    outlined_wrapper2() // for S2
  barrier2()
```

Select work posted by the master - the master changes what **work** points to.

Level1 - warp threads

```
#pragma omp target
{
  int A;
  #pragma omp parallel
  {
     int B;
     // S1 using A, B
     #pragma omp parallel
     {
        // S2 using B
     }
  }
}
```



```
kernel_workers():
  barrier1()
  if (work == 1)
    outlined_wrapper1() // for S1
  if (work == 2)
    outlined_wrapper2() // for S2
  barrier2()
```

```
outlined_wrapper1():
  Level0 *S = call_to_runtime_to_get_S();
  ptr_A = &S->A;
  outlined1(ptr_A)
```

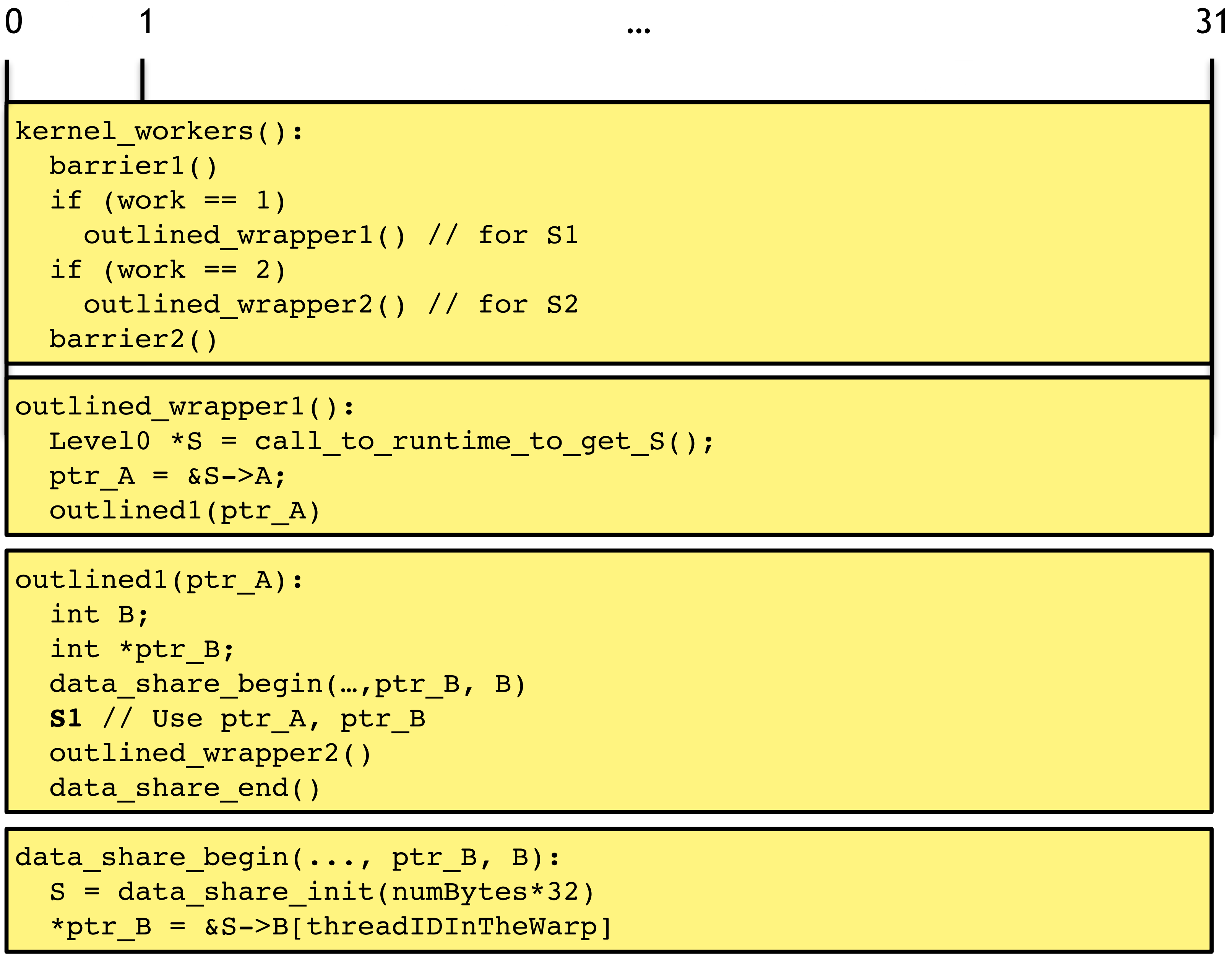
Select work posted by the master - the master changes what **work** points to.

Level1 - warp threads

```
#pragma omp target
{
  int A;
  #pragma omp parallel
  {
     int B;
     // S1 using A, B
     #pragma omp parallel
     {
        // S2 using B
     }
  }
}
```

S1 is executing using **ptr_A** to shared value of **A**.

Each thread in the warp manages its own value of **B**. Accesses through **ptr_B** are **coalesced**.



```
kernel_workers():
  barrier1()
  if (work == 1)
    outlined_wrapper1() // for S1
  if (work == 2)
    outlined_wrapper2() // for S2
  barrier2()
```

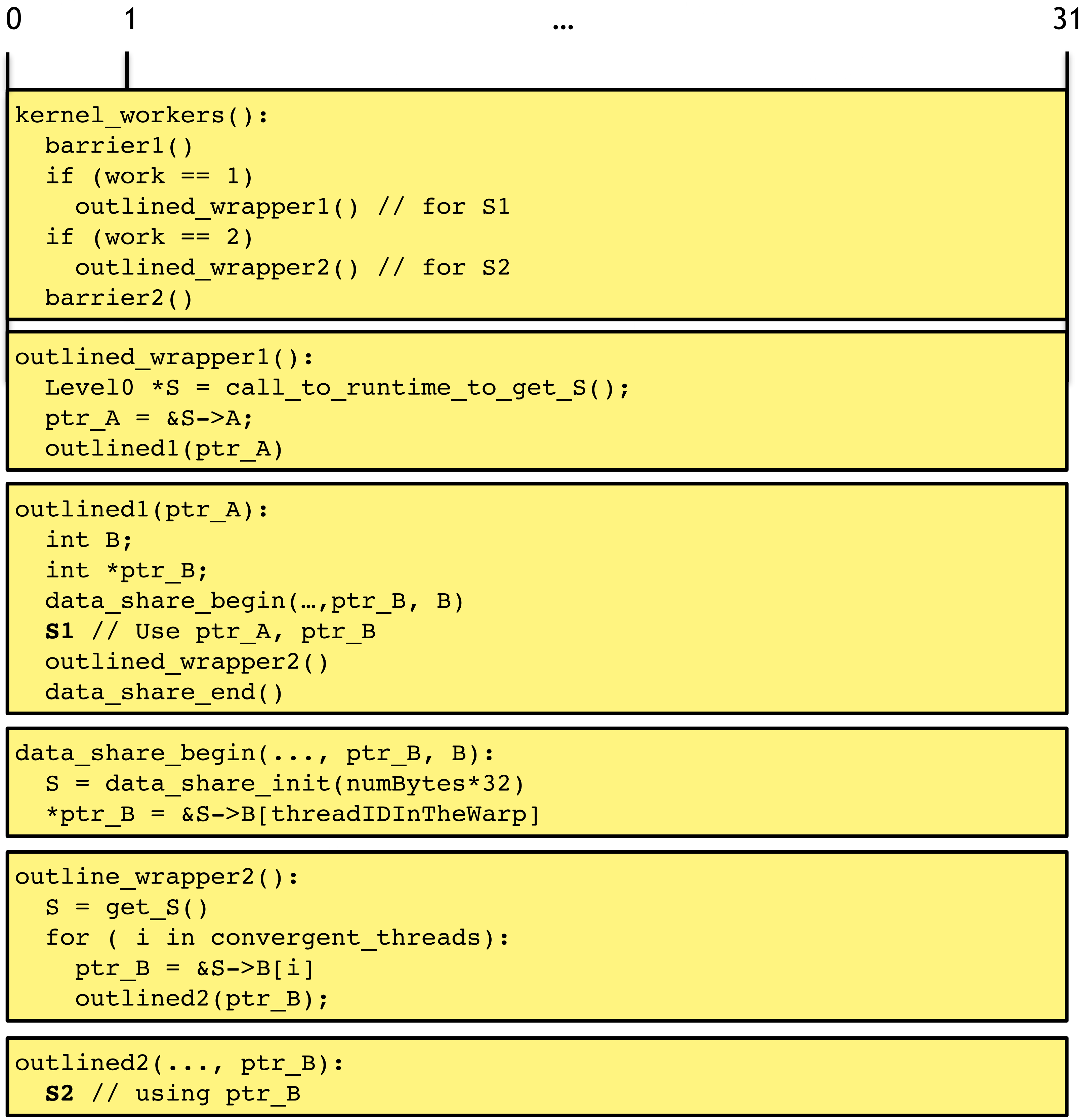
```
outlined_wrapper1():
  Level0 *S = call_to_runtime_to_get_S();
  ptr_A = &S->A;
  outlined1(ptr_A)
```

```
outlined1(ptr_A):
  int B;
  int *ptr_B;
  data_share_begin(…,ptr_B, B)
  S1 // Use ptr_A, ptr_B
  outlined_wrapper2()
  data_share_end()
```

```
data_share_begin(..., ptr_B, B):
  S = data_share_init(numBytes*32)
  *ptr_B = &S->B[threadIDInTheWarp]
```

Level1 - warp threads

```
#pragma omp target
{
   int A;
   #pragma omp parallel
   {
       int B;
       // S1 using A, B
       #pragma omp parallel
       {
           // S2 using B
       }
   }
}
```



```
kernel_workers():
  barrier1()
  if (work == 1)
    outlined_wrapper1() // for S1
  if (work == 2)
    outlined_wrapper2() // for S2
  barrier2()
```

```
outlined_wrapper1():
  Level0 *S = call_to_runtime_to_get_S();
  ptr_A = &S->A;
  outlined1(ptr_A)
```

```
outlined1(ptr_A):
  int B;
  int *ptr_B;
  data_share_begin(…,ptr_B, B)
  S1 // Use ptr_A, ptr_B
  outlined_wrapper2()
  data_share_end()
```

```
data_share_begin(..., ptr_B, B):
  S = data_share_init(numBytes*32)
  *ptr_B = &S->B[threadIDInTheWarp]
```

```
outline_wrapper2():
  S = get_S()
  for ( i in convergent_threads):
    ptr_B = &S->B[i]
    outlined2(ptr_B);
```

```
outlined2(..., ptr_B):
  S2 // using ptr_B
```

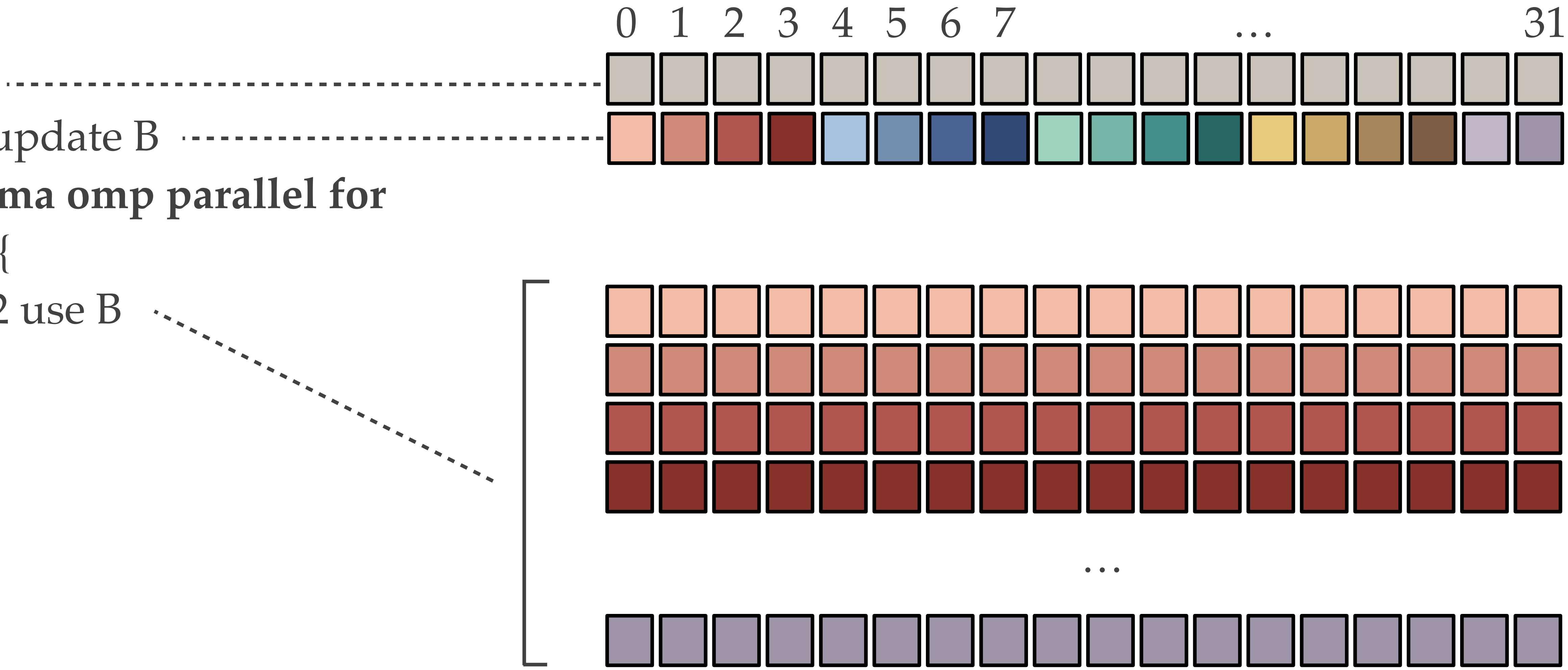
The work is shared between the active threads in the warp.

Level1 data sharing

Instances of **B** for kernel launched with **1 team** and **128 threads per team**

```
#pragma omp target
{
  #pragma omp parallel for
  for(…) {
    int B;
    // S1 update B
    # pragma omp parallel for
    for(…){
      // S2 use B
    }
  }
}
```

Instances of B for 1 warp



Total data sharing volumes

```
int A;
```

- Kernel is launched with **512 teams** and **1024 threads** per team (**32 warps** per team).

- Depending on how **A** is shared we can compute the **total memory size required for all instances of A:**
  - **A** is a global value shared amongst all teams: **4 B**
  - **A** is a value shared from the team master to the rest of the threads in a team: 4 x 512 = 2048 B = **2 KB**
  - **A** is a global value shared among all threads within a warp: 4 x 512 x 32 = **64 KB**

Data sharing on a GPU SM

```
int A;
```

- Kernel is launched with **512 teams** and **1024 threads per team** (**32 warps** per team).
- Only some of the teams are running at a given time. There is a maximum of 2048 threads per SM when only 32 registers per thread are used. This means **2 teams can fit on an SM** at a given time.
- The K40 has **14 SMs**.

- The volume per GPU SM is given by:
  - **A** is a global value shared amongst all teams: **4 B**
  - **A** is shared from the team master to the rest of the threads in a team: 4 x 2 = **8 B**
  - **A** is a global value shared among all threads within a warp: 4 x 2 x 32 = **256 B**

Using Level0 and Level1 data sharing

```
int A;
```

- Kernel is launched with **512 teams** and **1024 threads** per team (**32 warps** per team).
- Only some of the teams are running at a given time. There is a maximum of 2048 threads per SM when only 32 registers per thread are used. This means **2 teams can fit on an SM** at a given time.
- The K40 has **14 SMs**.

- The shared data volume per SM is given by:
  - **A** is a global value shared amongst all teams: **4 B**
  - **Level0: A** is shared from the team master to the rest of the threads in a team: 4 x 2 = **8 B**
  - **Level1: A** is a global value shared among all threads within a warp. Since we are executing the region belonging to Level0 with all threads in the warp, we have a distinct value for **A** for each thread. The total volume per warp in this case is: 4 x 32 = **128 B.** Since we have 32 warps per team and 2 teams the volume of the shared instances of **A** is: 128 x 2 x 32 = **8192 B = 8 KB.**