IBM TJ Watson Research Center - Advanced Compiler Technologies

Implementing implicit OpenMP data sharing on GPUs

Gheorghe-Teodor Bercea

IBM Research

Team:
Gheorghe-Teodor (Doru) Bercea, Carlo Bertolli, Hyojin Sung, Arpith C. Jacob, Alexandre Eichenberger, Georgios Rokos, Alexey Bataev, Tong Chen, Kevin O'Brien.

SC17 Denver, CO hpc connects.

Overview

- Introducing an “upstream-able” data sharing scheme for CLANG/LLVM trunk.
- We cover only the **first level of sharing**: from one thread to the rest of the threads in the same OpenMP team.
- Overcoming the problem that:

“In certain use cases, OpenMP’s **default sharing of local variables** is incompatible with the **default allocation into local memory** of local variables on NVIDIA GPUs.”

Mapping OpenMP to GPUs

IBM

```
void test(){
  int c = 5000;
  #pragma omp target
  {
    c += 1;
    #pragma omp parallel for
    for (i) {
      A[i] = c * i;
    }
  }
}
```

OpenMP allows nesting of regions with different numbers of threads.

Mapping OpenMP to GPUs

IBM

OpenMP semantics

```
void test(){
  int c = 5000;
  #pragma omp target
  {
    c += 1;                  1 thread
    #pragma omp parallel for
    for (i) {
      A[i] = c * i;          all threads
    }
  }
}
```

Mapping OpenMP to GPUs

IBM

We need to share "c"

```
void test(){
  int c = 5000;
  #pragma omp target
  {
   c += 1;                    1 thread
    #pragma omp parallel for
    for (i) {
      A[i] = c * i;          all threads
    }
  }
}
```

Mapping OpenMP to GPUs

IBM

Default NVPTX backend policy: "c" is allocated onto the thread local stack

```
void test(){
  int c = 5000;
  #pragma omp target
  {
   c += 1;                    1 thread
    #pragma omp parallel for
    for (i) {
      A[i] = c * i;           all threads
    }
  }
}
```

Mapping OpenMP to GPUs

IBM

```
void test(){
   int c = 5000;
   #pragma omp target
   {
    c += 1;                    1 thread
     #pragma omp parallel for
     for (i) {
       A[i] = c * i;           all threads
     }
   }
}
```

Default NVPTX backend policy: "c" is allocated onto the thread local stack

On GPUs threads cannot share a variable allocated on the local stack.

Function outlining

- In general: **OpenMP regions** delimited by different constructs will be outlined.
- The master thread assigns those regions to workers **dynamically: we therefore avoid dynamic thread launch in favor of dynamic work allocation to existing threads**.
- Outlining ensures that all parallel OpenMP regions have access to all the worker threads including OpenMP regions that are defined in other compilation units.
- **Data must be shared across multiple functions.**

OpenMP outlined regions example

```
void test(){
  int c = 5000;
  #pragma omp target
  {
    c += 1;                      MASTER
    #pragma omp parallel for
    for (i) {
      A[i] = c * i;              WORKERS
    }
    c += 2;                      MASTER
  }
}
```

Changes to CLANG and the runtime

- The runtime maintains a list of references to the shared variables.
- The MASTER region needs to initialize this list.
- The WORKER region retrieves the list from the runtime and passes the arguments to the outlined parallel region (in the expected order).

Code generation for MASTER (1)

IBM

```
define void @KERNEL(i32* dereferenceable(4) %c){
entry:
  %c.addr = alloca i32*, align 8
  %shared_args = alloca i8**, align 4
  br i1 %1, label %.worker, label %.mastercheck
.worker:
  call void @WORKER()
  br label %.exit
.mastercheck:
  br i1 %5, label %.master, label %.exit
.master:
… [only master thread left]
```

LLVM-IR

Code generation for MASTER (2)

IBM

```
…
.master:
  call void @__kmpc_kernel_init(i32 %thread_limit6)
  call void @__kmpc_kernel_prepare_parallel(
    […], i8*** %shared_args, i32 1)
  %17 = load i8**, i8*** %shared_args, align 8
  %22 = getelementptr inbounds i8*, i8** %17, i64 0
  %23 = bitcast i32* %1 to i8*
  store i8* %23, i8** %22, align 8
  call void @llvm.nvvm.barrier0()
  call void @llvm.nvvm.barrier0()
  …
```

LLVM-IR

Code generation for WORKER (1)

```
define void @WORKER(i32* dereferenceable(4) %c){
entry:
  %shared_args = alloca i8**, align 8
  br label %.await.work
.await.work:
  call void @llvm.nvvm.barrier0()
  %0 = call i1 @__kmpc_kernel_parallel(
    i8** %work_fn, i8*** %shared_args)
.execute.parallel:
  %5 = load i8**, i8*** %shared_args, align 8
  call void @__omp_outlined___wrapper(
    i16 0, i32 %master_tid, i8** %5)
```

LLVM-IR

Code generation for WORKER (2)

IBM

```
define void @__omp_outlined___wrapper(…, i8**){
entry:
  %c.addr = alloca i32*, align 8
  store i8** %2, i8*** %.addr2, align 8
next:
  %3 = load i8**, i8*** %.addr2, align 8
  %10 = getelementptr inbounds i8*, i8** %3, i64 0
  %11 = bitcast i8** %10 to i32**
  %12 = load i32*, i32** %11, align 8
  call void @__omp_outlined(…, i32* %12)
  ret void
}
```

LLVM-IR

Mapping OpenMP to GPUs

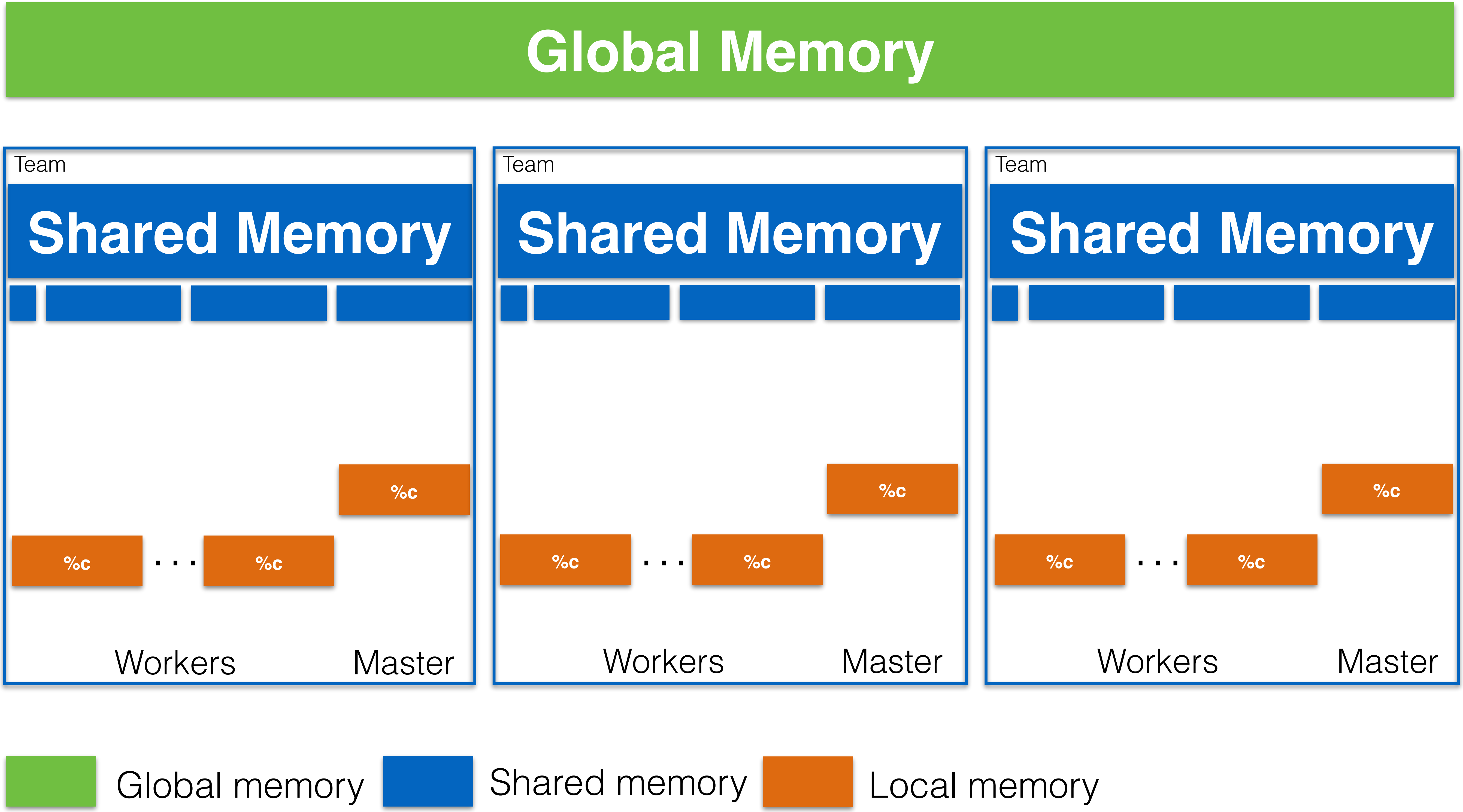
IBM

allocated in the MASTER thread's local memory by default, BUT must now be "shareable" with the WORKERS!

```
void test(){
  int c = 5000;
  #pragma omp target
  {
    c += 1; // LLVM-IR: %c = alloca i32
    #pragma omp parallel for
    for (i) {
      A[i] = c * i;
    }
    c += 2;
  }
}
```

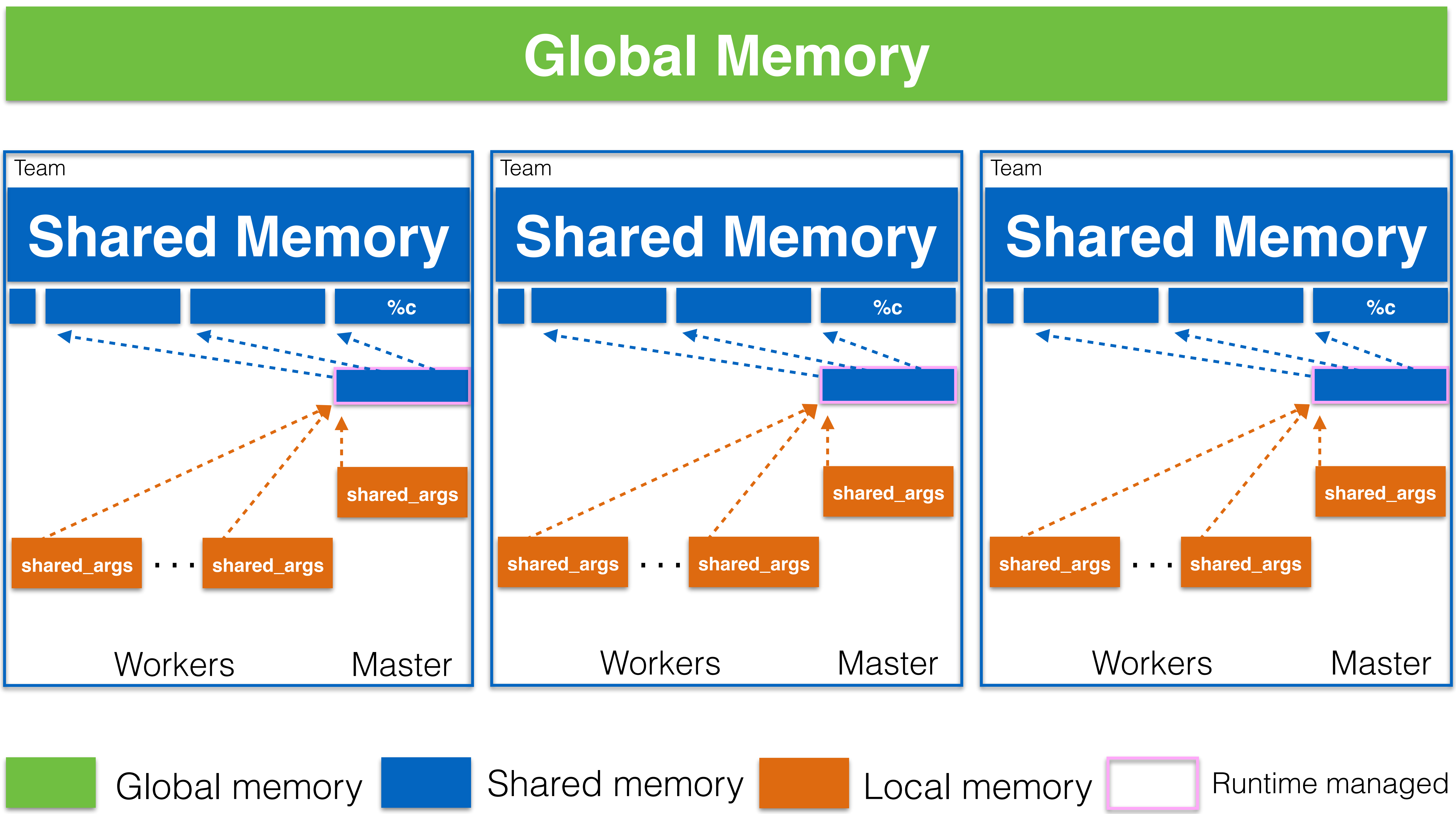
1. In the CUDA model shared variables must be explicitly declared as __shared__.
2. On a GPU, variables allocated in local memory cannot be shared.

1. No Sharing



Global Memory
Team
Shared Memory
%c
%c
. . .
%c
Workers
Master
Team
Shared Memory
%c
%c
. . .
%c
Workers
Master
Team
Shared Memory
%c
%c
. . .
%c
Workers
Master
Global memory
Shared memory
Local memory

2. Use device shared memory



Detecting implicitly shared variables

IBM

- Detecting shared variables:
  - Since sharing is supposed to happen implicitly, we need to detect the situation in which a variable is shared.
  - A variable is considered shared if its address is stored.
  - Avoids passing data from CLANG to LLVM backend about which variables are shared.
  - Limitation: too conservative, might end up sharing more than needed.

Local stack

- Currently only a local stack is used which resides in the prolog of the the function. It uses:
  - **SP** for **generic** address space operations.
  - **SPL** for **local** address space operations.

```
kernel() {
 .local  .align 8 .b8 __local_depot[10]

 mov.u64             %SPL, __local_depot
 cvta.local.u64  %SP, %SPL

 add.u64             %rd1, %SPL, 8
 ld.local.u64    %rd2, [%rd1]
 …
}
```

PTX

Add a shared stack

- Extend lowering of **alloca**'s to shared memory using **SPSH** for **shared** address space operations.

```
kernel() {
 .local  .align 8 .b8 __local_depot[10]
 .shared .align 8 .b8 __shared_depot[10]

 mov.u64          %SPL, __local_depot
 mov.u64          %SPSH, __shared_depot
 cvta.local.u64   %SP, %SPL
 cvta.shared.u64  %SP, %SPSH

 add.u64          %rd1, %SPSH, 8
 ld.shared.u64    %rd2, [%rd1]
 …
}
```

PTX

NVPTX backend passes

IBM

- **LowerSharedFrameIndices** (new pass for all optimization levels):
  - For -O0 insert before stack slot allocation.
  - For -O1 or higher insert before StackColoring pass:
    - ensures correctness of the stack slot coloring algorithm. Without this, the same local stack slot may be used by both a local and a shared variable. The StackColoring pass works on frame indices only.
  - Lowers frame indices to use the shared stack pointer **SPSH**.
  - Limitation: uses the same offsets as the local stack frame hence the shared and local stack frames have the same size.
  - Only lowers frame indices which fulfill the following condition:

```
%vreg25<def> = LEA_ADDRi64 <fi#3>, 0;
%vreg6<def> = cvta_to_shared_yes_64 %vreg25<kill>;
```

MI

```
%vreg25<def> = LEA_ADDRi64 %VRShared, 32;
```

MI

MI = Machine Instruction

NVPTX backend passes

IBM

- **LowerAlloca** (for -O1 or higher):
  - **Currently**: inserts instructions to that convert between the **generic** and **local** address spaces.
  - **Add**: conversion between **generic** and **shared** address spaces - the decision to lower to different address spaces needs to happen at the same time for all address spaces.
- **FunctionDataSharing** (New pass for -O0):
  - conversion between **generic** and **shared** address spaces
- The **NVPTXInferAddressSpaces** will do the actual lowering by coupling last two instructions

```
%A = alloca i32
store i32 0, i32* %A ; emits st.u32
```

LLVM-IR

```
%A = alloca i32
%Shared = addrspacecast i32* %A to i32 addrspace(3)*
%Generic = addrspacecast i32 addrspace(3)* %A to i32*
; the following instruction emits: st.shared.u32
store i32 0, i32 addrspace(3)* %Generic
```

LLVM-IR

Performance - data volume

IBM

Number of variables	Static shared memory per team [Bytes]	Dynamic global memory per team [Bytes]	Registers	Teams/SM	Shared memory per SM [Bytes]
1	233	0	36	14	3262
2	241	0	36	14	3374
4	257	0	36	14	3598
8	289	0	36	14	4046
16	353	0	40	12	4236
32	481	256	72	7	3367
64	737	512	136	3	2211

- When sharing variables, the shared memory volume that the scheme requires is relatively low.
- In most cases register usage becomes a problem before data sharing does.

Performance - data volume

Number of variables	Array data shared [Bytes]	Static shared memory per team [Bytes]	Registers	Potential Teams/SM	Shared memory per SM [Bytes]	Actual Teams/SM
1	384	617	36	14	8638	14
2	768	1001	36	14	14014	14
3	1152	1385	36	14	19390	11
4	1536	1769	36	14	24766	9

- Sharing arrays does not increase register pressure.
- Shared memory usage can limit occupancy in this case.
- Shared memory is not enough …

Limitations & future work

- Limitations of the **new** data sharing scheme:
  - **No communication from CLANG to LLVM about OpenMP:** CUDA and OpenMP offloading share the same toolchain, distinguish between the two.
  - **Shared memory is limited:** adopt one of the more generic sharing alternatives for cases in which shared memory is insufficient or inefficient due to occupancy.
  - **Support for recursive functions**
  - **Support second level of sharing among WORKERS:** currently the new data sharing infrastructure only supports sharing from MASTER to WORKERS.
- These limitations do not apply to the current data sharing scheme.

Future work: sharing among workers

```
void test(){
  int c = 5000;
  #pragma omp target
  {
    c += 1;
    #pragma omp parallel for
    for (i) {
      int d;
      d = c * i;
      #pragma omp simd
      for (j) {
         B[j] = d * j;
      }
    }
    c += 2;
  }
}
```

Putting it all together

- Addition of a shared memory scheme compatible with the current code generation scheme:
  - we modified the runtime to share values from MASTER to WORKER threads.
  - we modified CLANG's code generation to support our data sharing convention.
- Sharing relies on variables being stored in a "shareable" memory address space on the device:
  - we modified LLVM's NVPTX Backend to support the lowering of shared variables to the GPU's shared memory.

Thank you for listening!

Questions?

Changes to LLVM's NVPTX Backend

- There are 4 alternative ways for lowering a shared variable:
  1. lower alloca to local memory - no sharing needed;
  2. lower alloca to shared memory - one instance of the shared variable per team, store variable in shared memory stack, limited by shared memory size;
  3. lower alloca to global memory - one instance per team but in global memory, no more team-level management of the variable, vulnerable to recursive functions;
  4. lower alloca to runtime-managed memory - use a global memory stack managed by the runtime, supports all cases, interactions with runtime are expensive.