OpenMP

Stony Brook University AMD Lawrence Livermore National Laboratory

Direct GPU Compilation and Execution for Host Applications with OpenMP Parallelism

Shilei Tian[1], Joseph Huber[2], Konstantinos Parasyris[3], Barbara Chapman[1], and Johannes Doerfert[3]

[1] Stony Brook University
[2] Advanced Micro Devices
[3] Lawrence Livermore National Laboratory

OpenMP

Stony Brook University AMD Lawrence Livermore National Laboratory

The views and opinions of the authors do not necessarily reflect those of the U.S. government or Lawrence Livermore National Security, LLC neither of whom nor any of their employees make any endorsements, express or implied warranties or representations or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of the information contained herein.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative. We also gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory. This work was partially prepared by LLNL under Contract DE-AC52-07NA27344 and was partially supported by the LLNL-LDRD Program under Project No. 21-ERD-018.

Host OpenMP Application

```
extern int foo(int a, int b);

int main(int argc, char *argv[]) {
  FILE *fp = fopen("data.txt", "r");
#pragma omp parallel for
  for (int i = 0; i < n; ++i)
    c[i] = foo(a[i], b[i]);
  return 0;
}
```

Port to CUDA

- kernel entry point
- device function
- index calculation
- memory mapping
- kernel launch

```
extern __device__ int foo(int a, int b);
__global__ void kernel(int *a, int *b, int *c) {
  int i = blockId.x * blockDim.x + threadId.x;
  c[i] = foo(a[i], b[i]);
}
int main(int argc, char *argv[]) {
  FILE *fp = fopen("data.txt", "r");
  int *da, *db, *dc;
  cudaMemcpy(da, a, ...);
  cudaMemcpy(db, b, ...);
  cudaMemcpy(dc, c, ...);
  kernel<<<...>>>(da, db, dc);
  return 0;
}
```

Port to OpenMP Offloading

- kernel entry point
- ~~device function~~
- ~~index calculation~~
- memory mapping
- kernel launch

```
extern int foo(int a, int b);

int main(int argc, char *argv[]) {
  FILE *fp = fopen("data.txt", "r");
#pragma omp target teams distribute parallel for\
        map(to: a[n], b[n]) map(from: c[n])
  for (int i = 0; i < n; ++i)
    c[i] = foo(a[i], b[i]);
  return 0;
}
```

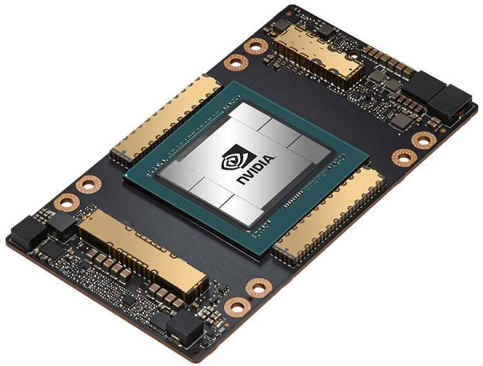
What If I Don’t Want to Port?

Can I just do something like…?

```
$ clang -f”run-on-gpu” my_app.c -o exec_on_gpu
```

and then

```
$ ./exec_on_gpu
```

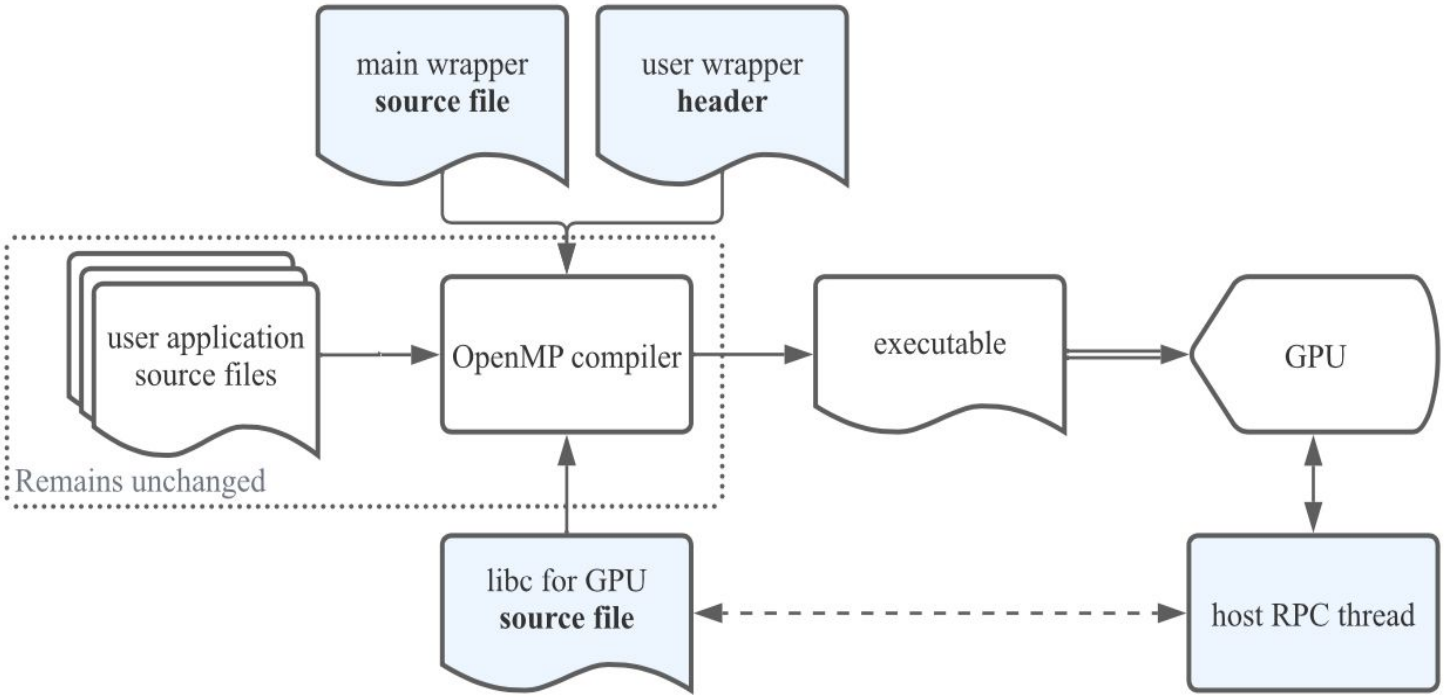


What Do We Need?

- device function
- kernel entry point
- ~~memory mapping~~
- ~~index calculation~~
- kernel launch
- library functions


```
extern int foo(int a, int b);

int main(int argc, char *argv[]) {
  FILE *fp = fopen("data.txt", "r");
#pragma omp parallel for
  for (int i = 0; i < n; ++i)
    c[i] = foo(a[i], b[i]);
  return 0;
}
```

Direct GPU Compilation



User Wrapper Header

```
#pragma omp begin declare target device_type(nohost)
int g;
void foo();
#pragma omp end declare target
```

User Wrapper Header

```
// UserWrapper.h
#pragma omp begin declare target device_type(nohost)
// <eof>
```

```
$ clang -include UserWrapper.h -c <user source files> ...
```

User Wrapper Header

```
#pragma omp begin declare target device_type(nohost)
extern int foo(int a, int b);

int main(int argc, char *argv[]) {
  FILE *fp = fopen("data.txt", "r");
#pragma omp parallel for
  for (int i = 0; i < n; ++i)
    c[i] = foo(a[i], b[i]);
  return 0;
}
#pragma omp end declare target
```

Main Wrapper

```
extern int __user_main(int, char *[]);

int main(int argc, char *argv[]) {
#pragma omp target enter data map(to: argv[:argc])

  for (int I = 0; I < argc; ++I) {
    size_t Len = strlen(argv[I]);
#pragma omp target enter data map(to: argv[I][:Len])
  }

  int Ret;
#pragma omp target teams num_teams(1) thread_limit(1024) map(from: Ret)
  { Ret = __user_main(argc, argv); }
  return Ret;
}
```

Kernel Entry Point

```
// UserWrapper.h
#pragma omp begin declare target device_type(nohost)

int main(int, char *[]) asm("__user_main");
// <eof>
```

Teams and num_teams(1)?

```
extern int __user_main(int, char *[]);

int main(int argc, char *argv[]) {
#pragma omp target enter data map(to: argv[:argc])

  for (int I = 0; I < argc; ++I) {
    size_t Len = strlen(argv[I]);
#pragma omp target enter data map(to: argv[I][:Len])
  }

  int Ret;
#pragma omp target teams num_teams(1) thread_limit(1024) map(from: Ret)
  { Ret = __user_main(argc, argv); }
  return Ret;
}
```

OpenMP Execution Model

```
void foo() {

  {
    /* region 1 */
#pragma omp parallel
    { /* region 2 */ }
    /* region 3 */
  }
}
```

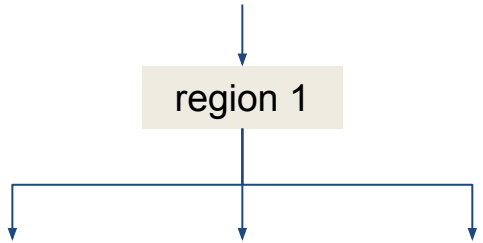
OpenMP Execution Model

```
void foo() {

  {
    /* region 1 */
#pragma omp parallel
    { /* region 2 */ }
    /* region 3 */
  }
}
```

region 1

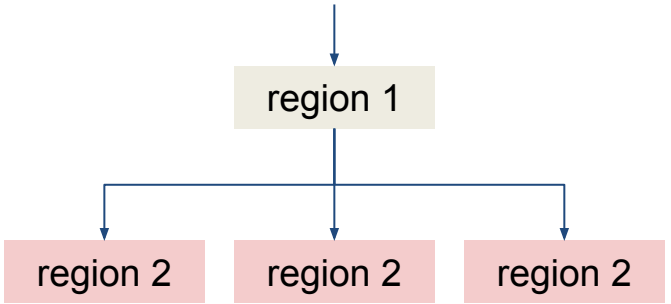
OpenMP Execution Model

```
void foo() {

  {
    /* region 1 */
#pragma omp parallel
    { /* region 2 */ }
    /* region 3 */
  }
}
```



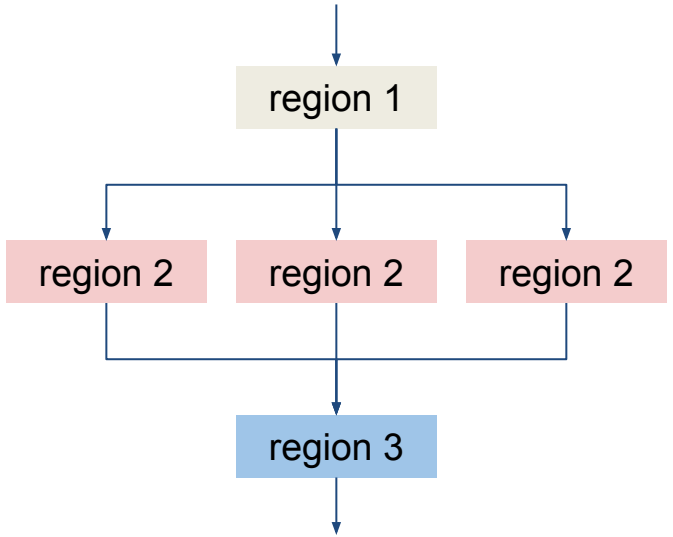
OpenMP Execution Model

```
void foo() {

  {
    /* region 1 */
#pragma omp parallel
    { /* region 2 */ }
    /* region 3 */
  }
}
```



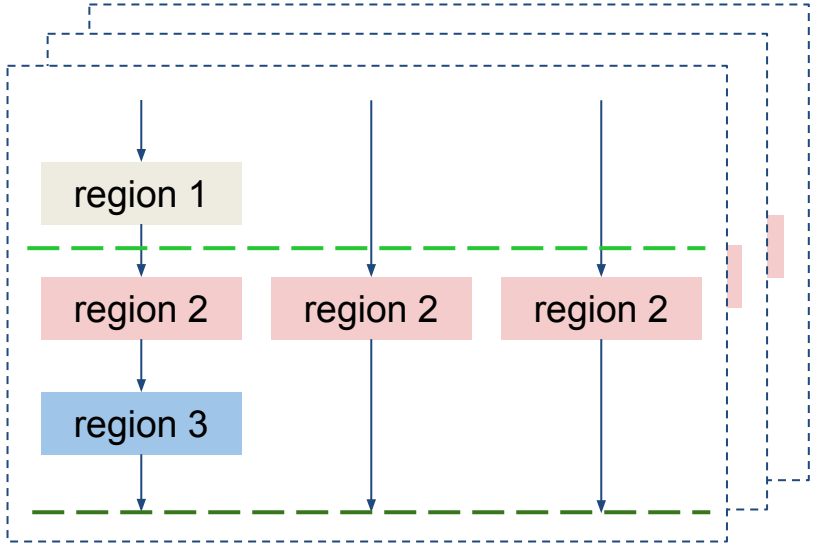
OpenMP Execution Model

```
void foo() {

   {
     /* region 1 */
#pragma omp parallel
     { /* region 2 */ }
     /* region 3 */
   }
}
```



OpenMP Execution Model

```
void foo() {
#pragma omp target teams num_teams(N)
   {
     /* region 1 */
#pragma omp parallel
     { /* region 2 */ }
     /* region 3 */
   }
}
```



Standard C library

1) Memory-related functionality, e.g., `malloc` and `free`
2) Utilities, such as `strcmp`, `atof`, `atoi`, and `memcpy`
3) I/O access via `fread`, `fprintf`, and similar functions

Standard C library

1) Memory-related functionality, e.g., `malloc` and `free`
   - GPU support varies among vendors
   - Implemented custom dynamic heap allocation
2) Utilities, such as `strcmp`, `atof`, `atoi`, and `memcpy`
3) I/O access via `fread`, `fprintf`, and similar functions

Standard C library

1) Memory-related functionality, e.g., `malloc` and `free`
2) Utilities, such as `strcmp`, `atof`, `atoi`, and `memcpy`
   - Implemented in a device library linked into the application
3) I/O access via `fread`, `fprintf`, and similar functions

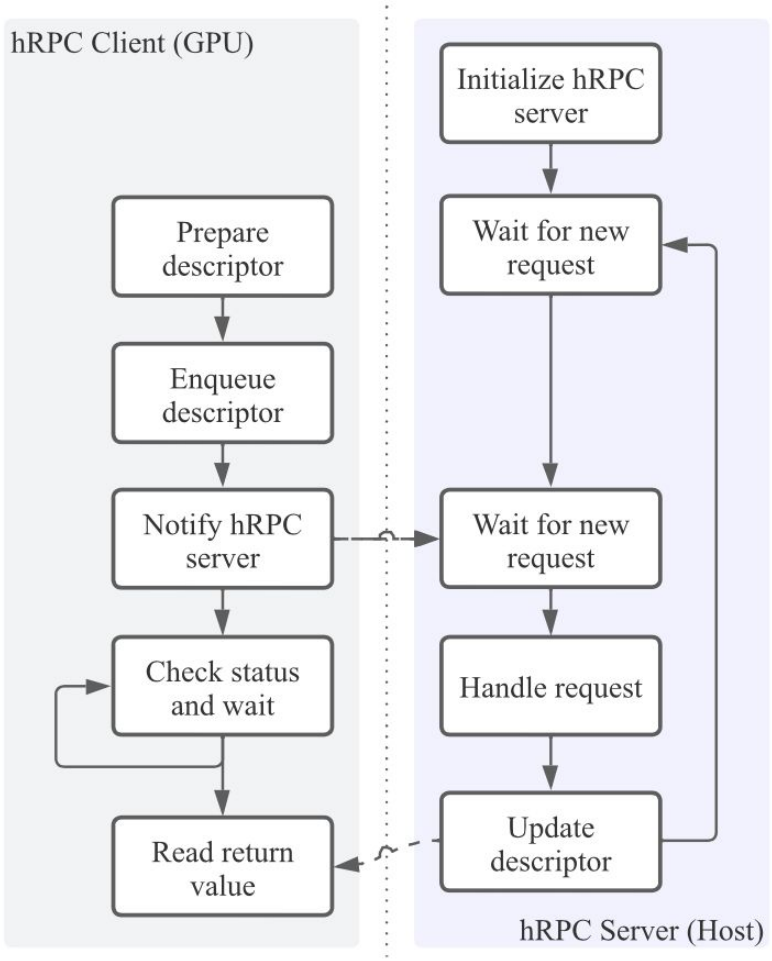
Standard C library

1) Memory-related functionality, e.g., `malloc` and `free`
2) Utilities, such as `strcmp`, `atof`, `atoi`, and `memcpy`
3) I/O access via `fread`, `fprintf`, and similar functions
   - Implemented via host remote procedure call (RPC)

Host RPC

Synchronous, stateless client-server protocol:

GPU (client) sends requests to host (server) and waits for completion



Example: Implement of fopen

```
FILE *fopen(const char *filename, const char *mode) {
  HostRPCDescriptorWrapper Wrapper(ID_fopen, 2);
  if (!Wrapper.isValid())
    return nullptr;

  auto Len1 = strlen(filename) + 1;
  auto Len2 = strlen(mode) + 1;

  HostRPCObject<const char *> FileName(Len1);
  HostRPCObject<const char *> Mode(Len2);

  FileName.copyFrom((void *)filename, Len1);
  Mode.copyFrom((void *)mode, Len2);

  Wrapper.addArg(FileName.get(), ARG_POINTER, Len1);
  Wrapper.addArg(Mode.get(), ARG_POINTER, Len2);

  if (!Wrapper.sendAndWait())
    return nullptr;
  return Wrapper.getReturnValue<FILE *>();
}
```

```
bool handle_fopen(HostRPCDescriptor &SD) {
  ArgumentExtractor AE(SD);

  auto *FileName = AE.getArg<const char *>(0);
  auto *Mode = AE.getArg<const char *>(1);

  FILE *F = fopen(FileName, Mode);
  if (F == nullptr)
    return false;

  SD.ReturnValue = (void *)F;
  return true;
}
```

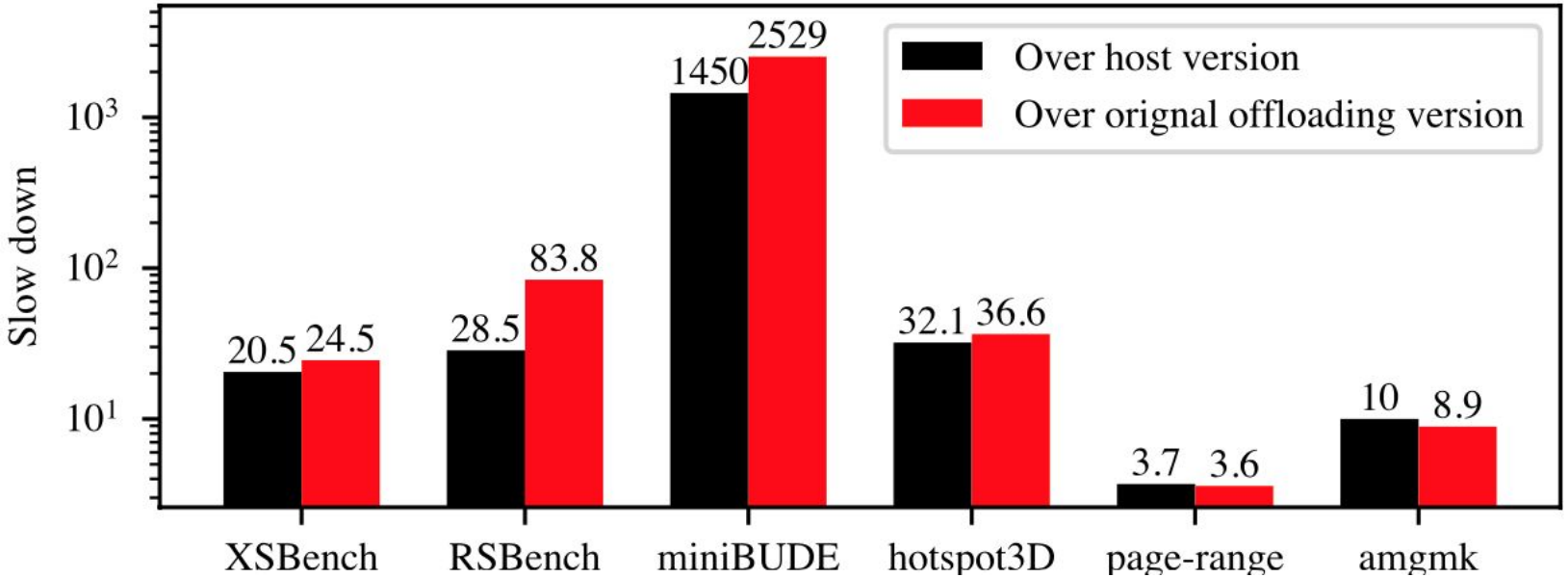
Putting Together

```
$ clang -include UserWrapper.h
        -fopenmp --offload-arch=<arch>
        -c <user source files>

$ clang -c <path to>/Main.c -o __Main.o
        -fopenmp --offload-arch=<arch> -fopenmp-offload-mandatory

$ clang -fopenmp --offload-arch=<arch>
        __Main.o <other object files>
        -o <exec name>
```

Limitations

- Arbitrary library functions, including C++ STL
- Variadic functions
- Single team execution

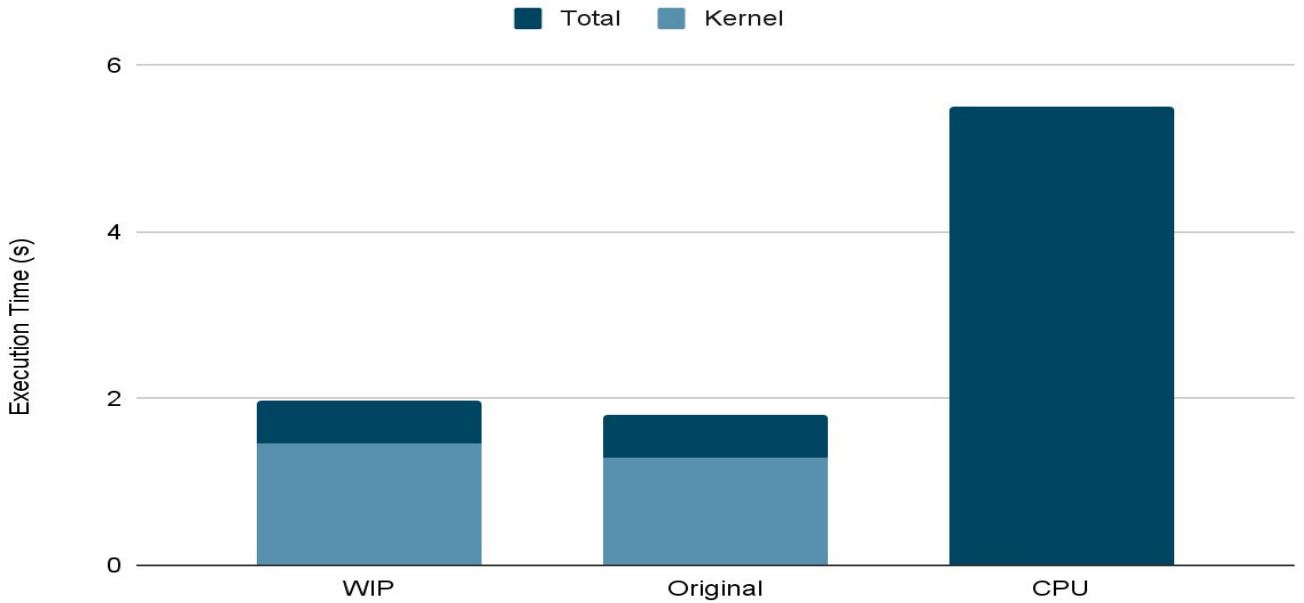
Performance Results



Lift the Limitations

- Support arbitrary library functions and variadic functions
  - Use new link time optimization pass
- Single team execution (performance issue)
  - Use multiple teams if a parallel region semantically allows it
    - XSBench can get back the performance of the actual computation part
  - If not, use “large team” that can have more than 1024 threads

New XSBench Result



Summary

- A user-transparent infrastructure that can compile the entire user program to have it directly running on GPUs.
  - Without the need to change the compiler.
- All the limitations and performance issues are being solved in our new prototype.

LibC for GPU

- LLVM libc officially supports GPU (partially yet)!
  - https://libc.llvm.org/gpu_mode.html
  - Basic device functions that do not require host are supported on the device.
    - `ctype.h` and `string.h`

Credit: Joseph Huber

OpenMP

Stony Brook University AMD Lawrence Livermore National Laboratory

FAQ

OpenMP

Stony Brook University AMD Lawrence Livermore National Laboratory

FAQ

What's the point of doing it?

OpenMP

Stony Brook University AMD Lawrence Livermore National Laboratory

FAQ

Why not just replacing pragmas?

OpenMP

Stony Brook University AMD Lawrence Livermore National Laboratory

FAQ

Do you support XXX?


THANK
YOU