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

2022 LLVM Developers' Meeting

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.

Port to GPU

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

int main(int argc, char *argv[]) {
#pragma omp parallel for
  for (int i = 0; i < n; ++i)
    c[i] = foo(a[i], b[i]);
  return 0;
}
```

Port to GPU

- 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[]) {
  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[]) {
#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;
}
```

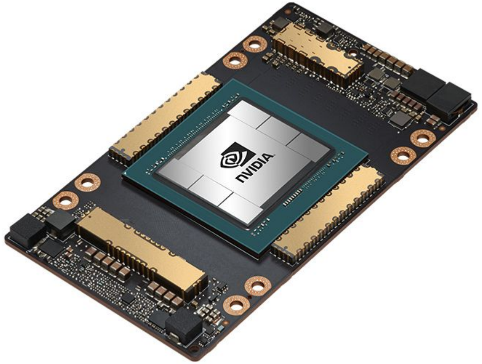
What If I Don’t Want to Port?

Can I just do something like…?

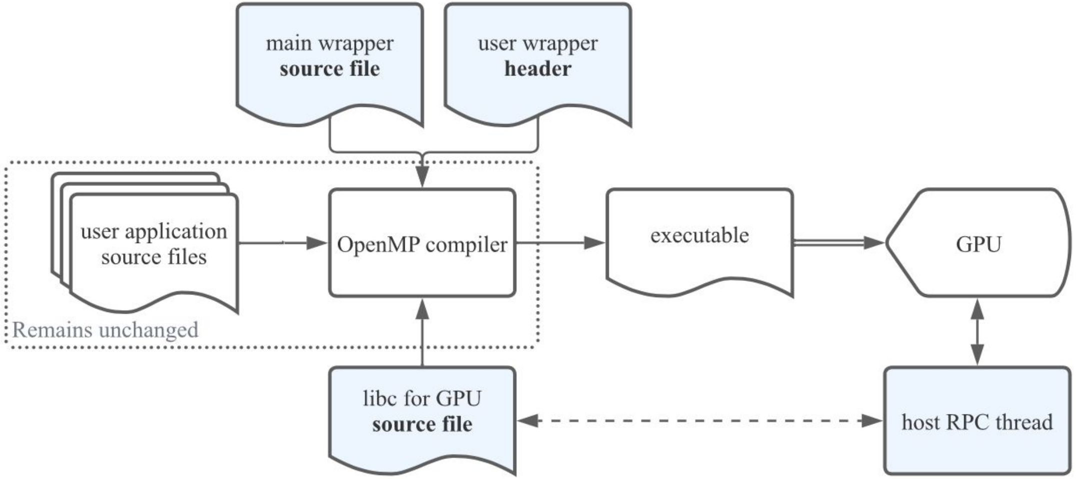
```
$ clang -f”run-on-gpu” my_app.c -o exec_on_gpu
```

and then

```
$ ./exec_on_gpu
```



Direct GPU Compilation



What Do We Need?

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

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

int main(int argc, char *argv[]) {
#pragma omp parallel for
  for (int i = 0; i < n; ++i)
    c[i] = foo(a[i], b[i]);
  return 0;
}
```

Device Function

```
#pragma omp begin declare target device_type(nohost)

int g;

void foo();

#pragma omp end declare target
```

Device Function

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

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

Device Function

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

int main(int argc, char *argv[]) {
#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
```

√ device function

- kernel launch
- kernel entry point

Kernel Launch

```
// Main.c
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 = main(argc, argv); }


  return Ret;
}
```

√ device function

√ kernel launch

- kernel entry point

Kernel Entry Point

```
// Main.c
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 = 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");
```

Kernel Entry Point

```
// Main.c
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;
}
```

√ device function

√ kernel launch

√ kernel entry point

Teams and num_teams(1)?

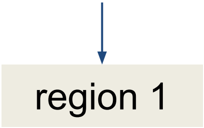
```
// Main.c
extern int __user_main(int, char *[]);

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 */
}
```

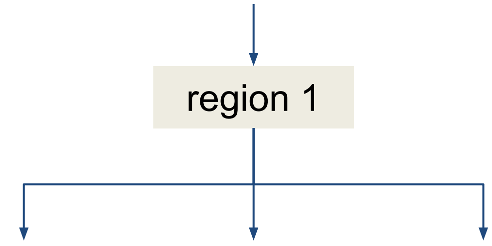
OpenMP Execution Model

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


region 1

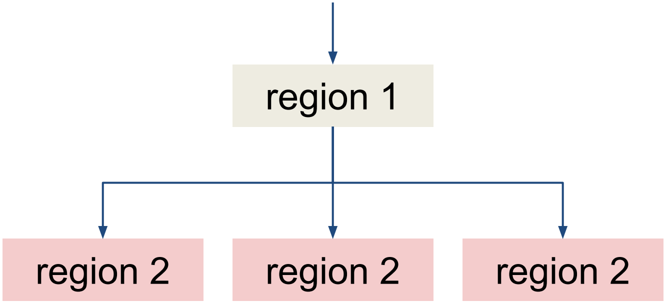
OpenMP Execution Model

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



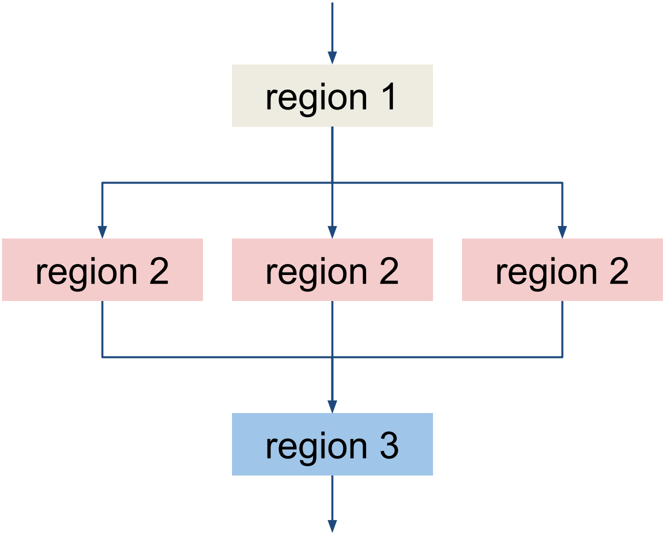
OpenMP Execution Model

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



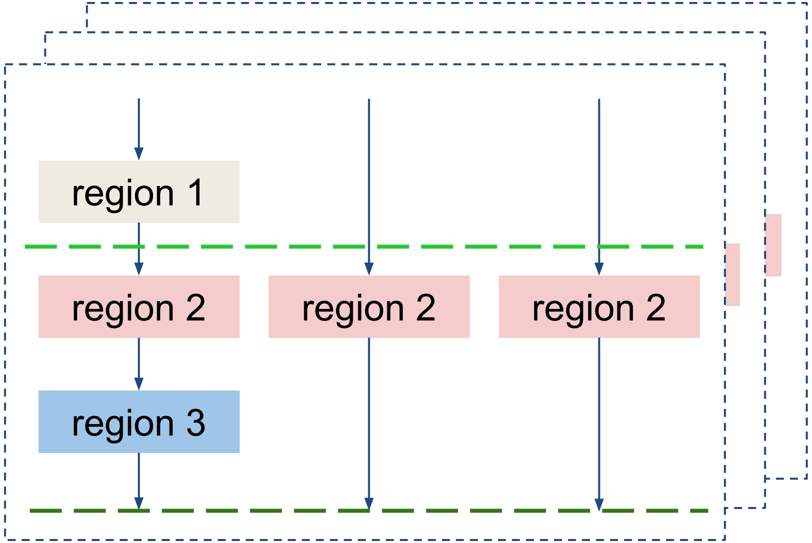
OpenMP Execution Model

```
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`
   - The support for GPU varies widely among vendors.
   - We implemented our own dynamic heap allocation.
2) Utilities, such as `strcmp`, `atof`, `atoi`, and `memcpy`
3) I/O access via `fread`, `fprintf`, and similar functions

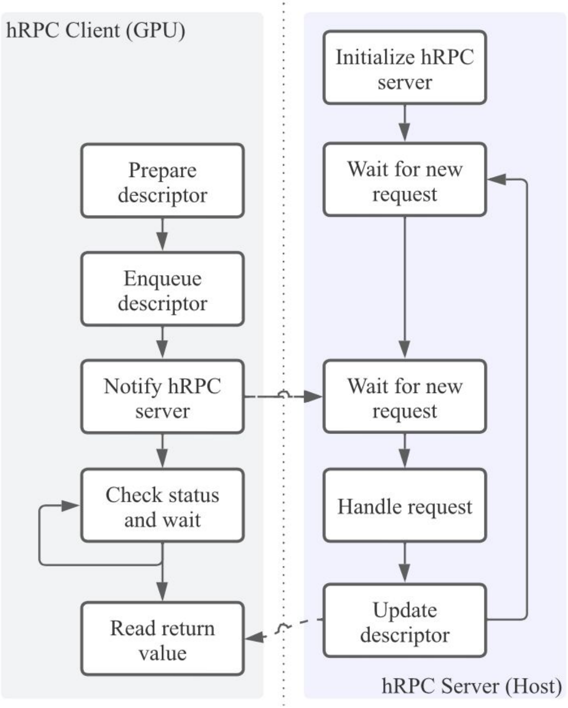
Standard C library

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

Standard C library

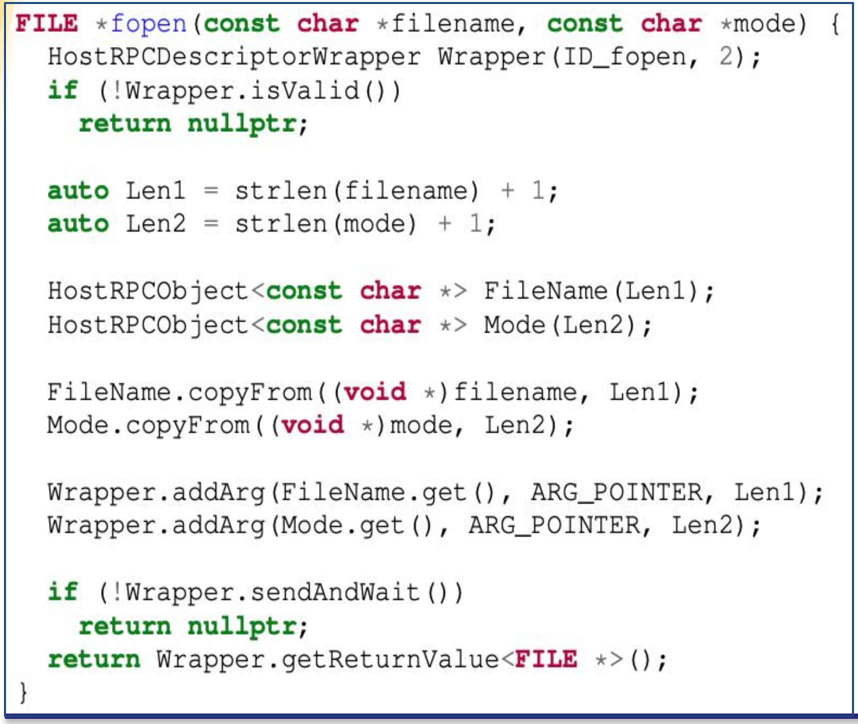
1) Memory-related functionality, e.g., `malloc` and `free`
2) Utilities, such as `strcmp`, `atof`, `atoi`, and `memcpy`
3) I/O access via `fread`, `fprintf`, and similar functions
    - They are implemented via host remote procedure call (RPC)

Host RPC

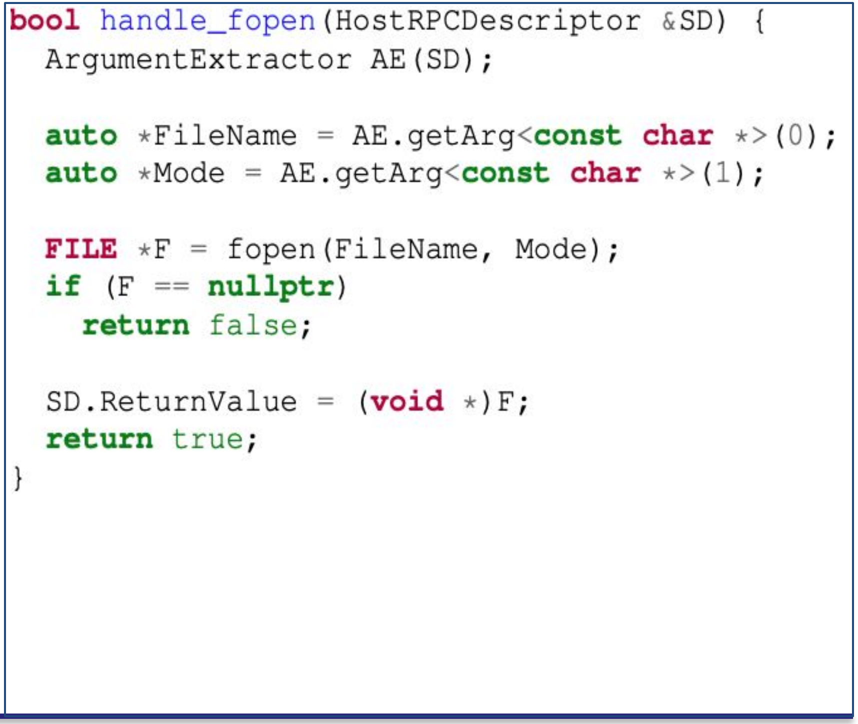


It features a synchronous, stateless client-server protocol, where the GPU (client) sends requests to the host (server) and waits for the host to acknowledge the completion.

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

$ 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>
```

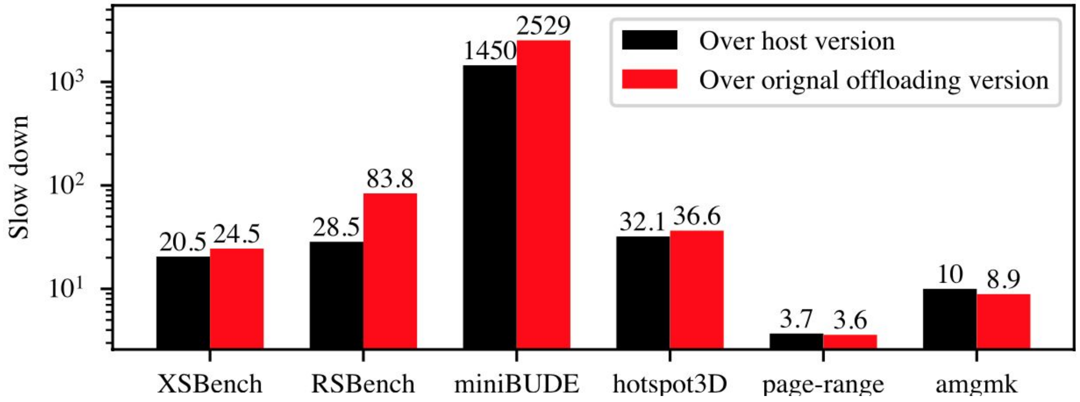
Limitations

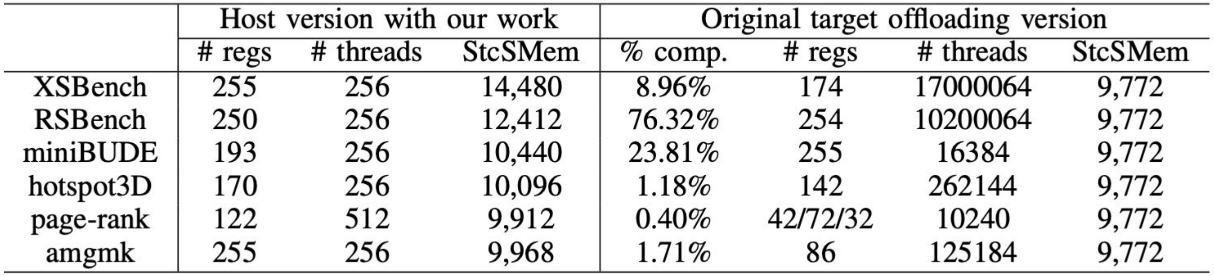
- Arbitrary library functions, including C++ STL
- Variadic functions
- Single team execution

Evaluation

- An Nvidia A100 GPU system with an AMD EPYC 7532 CPU (32C/64T) and 256 GB DDR4 RAM.
- Prototype version is based on 0a8dd8ef.
- Benchmarks:
  - XSBench and RSBench
  - miniBUDE
  - HeCBench
- Modify the code to work around the limitations.

Performance Results





	Host version with our work			Original target offloading version			
	# regs	# threads	StcSMem	% comp.	# regs	# threads	StcSMem
XSBench	255	256	14,480	8.96%	174	17000064	9,772
RSBench	250	256	12,412	76.32%	254	10200064	9,772
miniBUDE	193	256	10,440	23.81%	255	16384	9,772
hotspot3D	170	256	10,096	1.18%	142	262144	9,772
page-rank	122	512	9,912	0.40%	42/72/32	10240	9,772
amgmk	255	256	9,968	1.71%	86	125184	9,772

Future Work

- Support arbitrary library functions and variadic functions
  - Use compiler techniques to handle them.
- Single team execution
  - Use multiple teams if parallel regions semantically allow it.

Thanks

Port to GPU

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

int main(int argc, char *argv[]) {
#pragma omp parallel for
  for (int i = 0; i < n; ++i)
    c[i] = foo(a[i], b[i]);
  return 0;
}
```

```
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[]) {
  int *da, *db, *dc;
  cudaMemcpy(da, a, ...);
  cudaMemcpy(db, b, ...);
  cudaMemcpy(dc, c, ...);
  kernel<<<...>>>(da, db, dc);
  return 0;
}
```

Port to GPU

- kernel entry point

```
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[]) {
  int *da, *db, *dc;
  cudaMemcpy(da, a, ...);
  cudaMemcpy(db, b, ...);
  cudaMemcpy(dc, c, ...);
  kernel<<<...>>>(da, db, dc);
  return 0;
}
```

Port to GPU

- kernel entry point
- device function

```
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[]) {
  int *da, *db, *dc;
  cudaMemcpy(da, a, ...);
  cudaMemcpy(db, b, ...);
  cudaMemcpy(dc, c, ...);
  kernel<<<...>>>(da, db, dc);
  return 0;
}
```

Port to GPU

- kernel entry point
- device function
- index calculation

```
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[]) {
  int *da, *db, *dc;
  cudaMemcpy(da, a, ...);
  cudaMemcpy(db, b, ...);
  cudaMemcpy(dc, c, ...);
  kernel<<<...>>>(da, db, dc);
  return 0;
}
```

Port to GPU

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

```
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[]) {
  int *da, *db, *dc;
  cudaMemcpy(da, a, ...);
  cudaMemcpy(db, b, ...);
  cudaMemcpy(dc, c, ...);
  kernel<<<...>>>(da, db, dc);
  return 0;
}
```

Port to OpenMP Offloading

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

int main(int argc, char *argv[]) {
#pragma omp parallel for
  for (int i = 0; i < n; ++i)
    c[i] = foo(a[i], b[i]);
  return 0;
}
```

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

int main(int argc, char *argv[]) {
#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;
}
```