Developing an OpenMP Offloading Runtime for UVM-Capable GPUs

Hashim Sharif and Vikram Adve
University of Illinois at Urbana-Champaign

Hal Finkel
Argonne National Laboratory

Lingda Li
Brookhaven National Laboratory
Heterogenous Programming

➢ Heterogeneous programming allows for optimizing application subcomponents to their specific computation needs
Heterogeneous programming allows for optimizing application subcomponents to their specific computation needs

- OpenMP 4.0/4.5 supports heterogeneous programming
CUDA Unified Virtual Memory

➢ UVM provides a single memory space accessible by all GPUs and CPUs in the system

(CUDA 6 Unified Memory)

Kepler GPU

Unified Memory

CPU

(Limited to GPU Memory Size)
CUDA Unified Virtual Memory

- UVM provides a single memory space accessible by all GPUs and CPUs in the system
- Pascal introduces a page migration engine to enable automatic page migration on data access
CUDA Unified Virtual Memory

➢ UVM provides a single memory space accessible by all GPUs and CPUs in the system
➢ Pascal introduces a page migration engine to enable automatic page migration on data access
➢ Pascal UVM is only limited by the overall System Memory size
CUDA UVM Example

**CUDA Code**

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

    // Allocate memory for each vector on GPU
    cudaMalloc(&X, bytes);
    cudaMalloc(&Y, bytes);
    cudaMalloc(&Z, bytes);
    // Copy host vectors to device
    cudaMemcpy(x, X, bytes, hostToDevice);
    cudaMemcpy(y, Y, bytes, hostToDevice);
    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(X, Y, Z);
    // Copy the vector add result back to the host
    cudaMemcpy(z, Z, bytes, deviceToHost);
}
```

**CUDA UVM Code**

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

    // Allocate memory in Unified virtual space
    cudaMallocManaged(&X, bytes);
    cudaMallocManaged(&Y, bytes);
    cudaMallocManaged(&Z, bytes);
    // NOTE: No need for explicit memory copies
    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(X, Y, Z);
}
```
int main( int argc, char* argv[] ) {

    // Allocate memory for each vector on GPU
    cudaMalloc(&X, bytes);
    cudaMalloc(&Y, bytes);
    cudaMalloc(&Z, bytes);
    // Copy host vectors to device
    cudaMemcpy(x, X, bytes, hostToDevice);
    cudaMemcpy(y, Y, bytes, hostToDevice);
    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(X, Y, Z);
    // Copy the vector add result back to the host
    cudaMemcpy(z, Z, bytes, deviceToHost);
}

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

    // Allocate memory in Unified virtual space
    cudaMallocManaged(&X, bytes);
    cudaMallocManaged(&Y, bytes);
    cudaMallocManaged(&Z, bytes);
    // NOTE: No need for explicit memory copies
    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(X, Y, Z);
}

**CUDA UVM Example**

**CUDA Code**

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

    // Allocate memory for each vector on GPU
    cudaMalloc(&X, bytes);
    cudaMalloc(&Y, bytes);
    cudaMalloc(&Z, bytes);
    // Copy host vectors to device
    cudaMemcpy(x, X, bytes, hostToDevice);
    cudaMemcpy(y, Y, bytes, hostToDevice);
    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(X, Y, Z);
    // Copy the vector add result back to the host
    cudaMemcpy(z, Z, bytes, deviceToHost);
}
```

**CUDA UVM Code**

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

    // Allocate memory in Unified virtual space
    cudaMallocManaged(&X, bytes);
    cudaMallocManaged(&Y, bytes);
    cudaMallocManaged(&Z, bytes);
    // NOTE: No need for explicit memory copies
    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(X, Y, Z);
}
```
int main( int argc, char* argv[] ) {

    // Allocate memory for each vector on GPU
    cudaMalloc(&X, bytes);
    cudaMalloc(&Y, bytes);
    cudaMalloc(&Z, bytes);
    // Copy host vectors to device
    cudaMemcpy(x, X, bytes, hostToDevice);
    cudaMemcpy(y, Y, bytes, hostToDevice);
    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(X, Y, Z);
    // Copy the vector add result back to the host
    cudaMemcpy(z, Z, bytes, deviceToHost);
}

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

    // Allocate memory in Unified virtual space
    cudaMallocManaged(&X, bytes);
    cudaMallocManaged(&Y, bytes);
    cudaMallocManaged(&Z, bytes);
    // NOTE: No need for explicit memory copies
    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(X, Y, Z);
}
OpenMP Offloading
OpenMP Offloading

OpenMP threads share memory

Processor Y
- Cache Y

Processor X
- Cache X

RAM
- Shared Data
OpenMP Offloading

Processor Y
  ↓ Cache Y

RAM
  ↓ Shared Data

Processor X
  ↑ Cache X

Device Memory
  ↓ Bus

Accelerator
OpenMP Offloading

OpenMP 4 includes directives for mapping data to device memory.
OpenMP Target Directives - Saxpy

```c
#pragma omp target data map(to: x[0:N])
#pragma omp target data map(tofrom: y[0:N])
{
    #pragma omp target
    #pragma omp teams
    #pragma omp distribute parallel for
    for (long int i = 0; i < N; i++)
        y[i] = a*x[i] + y[i];
}
```
```c
#pragma omp target data map(to: x[0:N])
#pragma omp target data map(tofrom: y[0:N])
{
    #pragma omp target
    #pragma omp teams
    #pragma omp distribute parallel for
    for (long int i = 0; i < N; i++)
        y[i] = a * x[i] + y[i];
}
```

OpenMP Target Directives - Saxpy

directs target compilation
#pragma omp target data map(to: x[0:N])
#pragma omp target data map(tofrom: y[0:N])
{
    #pragma omp target
    #pragma omp teams
    #pragma omp distribute parallel for
    for (long int i = 0; i < N; i++)
        y[i] = a*x[i] + y[i];
}
OpenMP Offloading Runtime - libomptarget
OpenMP Offloading Runtime - libomptarget

Clang/LLVM Binary

- Non offloading constructs
- Target constructs

Host Code

Device Code

Host OpenMP RT

_libomp

Serves OpenMP Runtime calls for non-target regions
OpenMP Offloading Runtime - libomptarget

Clang/LLVM Binary

Host Code

Device Code

Non offloading constructs

Target constructs

Host OpenMP RT

libomp

Offloading Runtime

libomptarget

Device-agnostic openMP runtime; implementing support for target offload regions
OpenMP Offloading Runtime - libomptarget

- **Clang/LLVM Binary**
  - Host Code
  - Non offloading constructs
  - Target constructs
  - Device Code

- **Host OpenMP RT**
  - libomp

- **Offloading Runtime**
  - libomptarget

- **CUDA Device Plugin**
  - libomptarget

- **CUDA Driver API**

**Device plugins** implement device specific operations e.g. data copies, kernel execution
An OpenMP Framework for UVM

Goals

➢ Improved Performance
➢ Performance Portability

Design Considerations

➢ How can we leverage OpenMP target constructs?
➢ Does performance scale with large datasets?
An OpenMP Framework for UVM

Compiler Technology

➢ Developed as LLVM Transformations
➢ Extracts important application-specific information
  ➢ e.g data access probability

OpenMP Runtime

➢ Our implementation extends the *libomptarget* library
  ➢ More specifically, we extend the CUDA device plugin
  ➢ Developed a UVM-compatible offloading plugin
  ➢ Includes UVM-specific performance optimizations
Optimizations - Prefetching

```c
#pragma omp target data map(to: x[0:N])
#pragma omp target data map(tonfrom: y[0:N])
{
    #pragma omp target
    #pragma omp teams
    #pragma omp distribute parallel for
    for (long int i = 0; i < N; i++)
        y[i] = a*x[i] + y[i];
}
```
Optimizations - Prefetching

```c
#pragma omp target data map(to: x[0:N])
#pragma omp target data map(tofrom: y[0:N])
{
    #pragma omp target
    #pragma omp teams
    #pragma omp distribute parallel for
    for (long int i = 0; i < N; i++)
        y[i] = a * x[i] + y[i];
}
```

Data pages are fetched on demand. This leads to **page fault** processing overhead.
Optimizations - Prefetching

```c
#pragma omp target data map(to: x[0:N])
#pragma omp target data map(tofrom: y[0:N])
{
    #pragma omp target
    #pragma omp teams
    #pragma omp distribute parallel for
    for (long int i = 0; i < N; i++)
        y[i] = a*x[i] + y[i];
}
```

```
Runtime

PrefetchToDevice(x)
PrefetchToDevice(y)
ExecuteTargetRegion()
PrefetchToHost(y)
```
Optimizations - Prefetching

```c
#pragma omp target data map(to: x[0:N])
#pragma omp target data map(tofrom: y[0:N])
{
    #pragma omp target
    #pragma omp teams
    #pragma omp distribute parallel for
    for (long int i = 0; i < N; i++)
        y[i] = a*x[i] + y[i];
}
```

Prefetching used data pages helps avoid page fault processing overhead
Goal: The compiler extracts the access probabilities associated with the OpenMP mapped data
Goal: The compiler extracts the access probabilities associated with the OpenMP mapped data.
Prefetching Workflow - Compiler

**Goal:** The compiler extracts the access probabilities associated with the OpenMP mapped data.
Goal: The compiler extracts the access probabilities associated with the OpenMP mapped data
Prefetching Workflow - Runtime

Executable

API calls including access probabilities

Offloading Runtime - libomptarget
Prefetching Workflow - Runtime

Executable

API calls including access probabilities

Offloading Runtime - libomptarget

CUDA Device Plugin - libomptarget
Prefetching Workflow - Runtime

Executable

API calls including access probabilities

Offloading Runtime - libomptarget

CUDA Device Plugin - libomptarget

Uses a cost-model to determine the profitability of prefetching
Prefetching Workflow - Runtime

Executable

API calls including access probabilities

Offloading Runtime - libomptarget

CUDA Device Plugin - libomptarget

Uses a cost-model to determine the profitability of prefetching

Prefetch data with high access probability

Device Memory

Data Prefetching

RAM
Memory Oversubscription

- With device memory oversubscription, naïve data prefetching leads to memory thrashing
Memory Oversubscription

- With device memory oversubscription, naïve data prefetching leads to memory thrashing

- Pipelining partial prefetches with partial compute

(Now everything fits)
Pipelinining Workflow - Compiler
Pipelining Workflow - Compiler

App source ➔ Build ➔ LLVM IR

Compiler

LLVM Pass:
Extract OpenMP
Loop bounds
Pipelining Workflow - Compiler

App source -> LLVM IR

Compiler
LLVM Pass: Extract OpenMP Loop bounds

Required for chunking the iteration space
Pipelining Workflow - Compiler

App source \( \rightarrow \) LLVM IR \( \rightarrow \) Compiler

Compiler

- LLVM Pass: Extract OpenMP Loop bounds
- LLVM Pass: Extract memory access expressions

Required for chunking the iteration space
Pipelining Workflow - Compiler

App source → Build → LLVM IR

Compiler
LLVM Pass:
Extract OpenMP Loop bounds

LLVM Pass:
Extract memory access expressions

Required for chunking the iteration space
Required for chunking the data prefetches
Pipelining Workflow - Compiler

App source ➔ Build ➔ LLVM IR ➔ Compiler

LLVM IR

- Extract OpenMP Loop bounds
- Extract memory access expressions
- Add OpenMP Region Info

Transformed IR

- Required for chunking the iteration space
- Required for chunking the data prefetches
App source → LLVM IR → Compiler

Compiler:
- LLVM Pass: Extract OpenMP Loop bounds
- LLVM Pass: Extract memory access expressions
- LLVM Pass: Add OpenMP Region Info

Transformed IR → Binary

Required for chunking the iteration space
Required for chunking the data prefetches
Pipelining Workflow - Runtime

Application Binary

tgt_run_target()

Runtime API call including OpenMP Region Info

libomptarget

tgt_run_target()
Pipelining Workflow - Runtime

- Application Binary calls into the runtime to invoke the target offload region.

  - Application Binary calls `tgt_run_target()`.

- `tgt_run_target()` inside the runtime API call including OpenMP Region Info.

  - `tgt_run_target()` in `libomptarget`.

- `libomptarget` calls `tgt_run_target()`.
Pipelining Workflow - Runtime

Application Binary

tgt_run_target()

Runtime API call including OpenMP Region Info

libomptarget

tgt_run_target()

Device Plugin

chunkCount = getChunkCount()
for(i=0; i < chunkCount; i++){
    prefetchChunk()
    executeChunk()
    synchronizeTask()
}

Application calls into the runtime to invoke the target offload region
Pipelining Workflow - Runtime

Application Binary

```
tgt_run_target()
```

Runtime API call including OpenMP Region Info

Application calls into the runtime to invoke the target offload region

libomptarget

```
tgt_run_target()
```

Device Plugin

```
chunkCount = getChunkCount()
for(i=0; i < chunkCount; i++){
prefetchChunk()
executeChunk()
synchronizeTask()
}
```

The device plugins chunks a kernel into multiple smaller kernels – if the device memory is being oversubscribed
Pipelining Workflow - Runtime

Application Binary

```
tgt_run_target()
```

Runtime API call including OpenMP Region Info

Application calls into the runtime to invoke the target offload region

libomptarget

```
tgt_run_target()
```  

Device Plugin

```
chunkCount = getChunkCount()
for(i=0; i < chunkCount; i++){
    prefetchChunk()
    executeChunk()
    synchronizeTask()
}
```

The device plugins chunks a kernel into multiple smaller kernels – if the device memory is being oversubscribed
Pipelining Workflow - Runtime

Application Binary

tgt_run_target()

Runtime API call including OpenMP Region Info

libomptarget

tgt_run_target()

Device Plugin

chunkCount = getChunkCount()
for(i=0; i < chunkCount; i++){
    prefetchChunk()
    executeChunk()
    synchronizeTask()
}
Pipelining Workflow - Runtime

Application Binary

tgt_run_target()

Runtime API call including OpenMP Region Info

libomptarget

tgt_run_target()

Device Plugin

chunkCount = getChunkCount()
for (i=0; i < chunkCount; i++){
    prefetchChunk()
    executeChunk()
    synchronizeTask()
}

Application calls into the runtime to invoke the target offload region

The device plugins chunks a kernel into multiple smaller kernels – if the device memory is being oversubscribed
Pipelining Workflow - Runtime

Application Binary

tgt_run_target()

Runtime API call including OpenMP Region Info

libomptarget

tgt_run_target()

Device Plugin

chunkCount = getChunkCount()
for(i=0; i < chunkCount; i++){
    prefetchChunk()
    executeChunk()
    synchronizeTask()
}

The device plugins chunks a kernel into multiple smaller kernels – if the device memory is being oversubscribed

Application calls into the runtime to invoke the target offload region
Applying a runtime API call including OpenMP Region Info

**Pipelining Workflow - Runtime**

Application Binary

```
tgt_run_target()
```

Runtime API call including OpenMP Region Info

Libomptarget

```
tgt_run_target()
```

Device Plugin

```
chunkCount = getChunkCount()
for(i=0; i < chunkCount; i++){
  prefetchChunk()
  executeChunk()
}
synchronizeTask()
```

Application calls into the runtime to invoke the target offload region

The device plugins chunks a kernel into multiple smaller kernels – if the device memory is being oversubscribed
Experiments
Experiments

➢ Do the optimizations help improve performance for computational kernels?

➢ Do the optimizations scale with increasing dataset sizes?
Experiments

- **Baseline**: Demand Paging
- **Comparisons**: Prefetching, Pipelined Prefetching
- **Benchmarks**: Saxpy, Kmeans (Rodinia)
Experimental Setup

Hardware

- Summitdev cluster at ORNL
  - Pascal P100 GPU cards
  - 16GB device memory
  - Nvlink interconnect

Software

- Clang, LLVM
  - clang-ykt project
- libomp, libomptarget
  - clang-ykt project
Results - SAXPY

![Graph showing results of SAXPY with different log(N) elements (27, 28, 29, 30, 31) with axes for log(N) elements on the x-axis and Time (%) on the y-axis. The graph compares three methods: Paging, Prefetch, and Pipelining+Prefetching.]
Results - SAXPY

This does not fit in device memory
Kmeans – 10 Iterations

The diagram shows the time (in %) for different log(N) points (27, 28, 29, 30, 31) across three conditions: Paging, Prefetch, and Pipelining+Prefetching. The y-axis represents the time (%), and the x-axis represents the log(N) points.
Kmeans – 10 Iterations

This does not fit in device memory
Kmeans – 20 Iterations

![Bar chart showing time (%) for Paging, Prefetch, and Pipelining across log(N) points 27 to 31. The y-axis represents time in percentage, and the x-axis represents log(N) points. The chart indicates that Prefetch generally has the highest time percentage compared to Paging and Pipelining.](image-url)
Demand paging performs better – hardware learns to use a more suitable eviction policy.
Summary

➢ Developed an OpenMP Framework for UVM-capable GPUs

➢ Develop optimizations to reduce page fault processing overhead
  ➢ Prefetching
  ➢ Pipelining

➢ Optimizations enable reasonable improvements on benchmarks

➢ Future Work: Developing more sophisticated pipelining strategies
Thanks & Questions?
Acknowledgements

- The LLVM community (including our many contributing vendors)
- The SOLLVE project (part of the Exascale Computing Project)
- OLCF, ORNL for access to the SummitDev system.
- ALCF, ANL, and DOE
References

- [https://drive.google.com/file/d/0B-jX56_FbGKRM21sYlNYVnB4eFk/view](https://drive.google.com/file/d/0B-jX56_FbGKRM21sYlNYVnB4eFk/view)
- [https://www.nersc.gov/assets/Uploads/XE62011OpenMP.pdf](https://www.nersc.gov/assets/Uploads/XE62011OpenMP.pdf)
- [https://llvm-hpc3-workshop.github.io/slides/Bertolli.pdf](https://llvm-hpc3-workshop.github.io/slides/Bertolli.pdf)