GENERATING GPU KERNELS FROM CHAPEL'S FEATURES FOR PARALLELISM AND LOCALITY

Engin Kayraklioglu, Andy Stone, David Iten, Sarah Nguyen, Bradford L. Chamberlain, Michael Ferguson and Michelle Strout
Hewlett Packard Enterprise

SIAM PP22 - Code Generation and Transformation in HPC on Heterogeneous Platforms
February 26, 2022
WHAT IS CHAPEL?

Chapel: A modern parallel programming language

- portable & scalable
- open-source & collaborative

Goals:

- Support general parallel programming
- Make parallel programming at scale far more productive
CHAPEL BENCHMARKS TEND TO BE CONCISE, CLEAR, AND COMPETITIVE

STREAM TRIAD: C + MPI + OPENMP

use BlockDist;

config const m = 1000, alpha = 3.0;
const Dom = {1..n} dmapped ...
var A, B, C: [Dom] real;
B = 2.0;
C = 1.0;
A = B + alpha * C;

forall (_, r) in zip(Updates, RAStream()) do
  T[r & indexMask].xor(r);

HPCC RA: MPI KERNEL

The scalar equivalent is:

r = Ran;
if (Ran = (Ran << 1) ^ ((s64Int) Ran < ZERO64B ? POLY : ZERO64B);
else
  r = (Ran << 1) ^ (((s64Int) Ran < 0) ? POLY : 0);

BlockDist

config const m = 1000, alpha = 3.0;
const Dom = {1..n} dmapped ...
var A, B, C: [Dom] real;
B = 2.0;
C = 1.0;
A = B + alpha * C;

forall (_, r) in zip(Updates, RAStream()) do
  T[r & indexMask].xor(r);
CURRENT FLAGSHIP CHAPEL APPLICATIONS

**CHAMPS: 3D Unstructured CFD**
Éric Laurendeau, Simon Bourgault-Côté, Matthieu Parenteau, et al.
École Polytechnique Montréal

**ChplUltra: Simulating Ultralight Dark Matter**
Nikhil Padmanabhan, J. Luna Zagorac, et al.
Yale University / University of Auckland

**Arkouda: NumPy at Massive Scale**
Mike Merrill, Bill Reus, et al.
US DoD

**ChOp: Chapel-based Optimization**
Tiago Carneiro, Nouredine Melab, et al.
INRIA Lille, France

**CrayAI: Distributed Machine Learning**
Hewlett Packard Enterprise

Your application here?
GPU PROGRAMMING IN CHAPEL

Overview

• We are developing support for GPU programming in Chapel
  • GPUs are very common, yet challenging to program
  • GPU support is frequently asked about by users
  • it would improve upon Chapel’s “any parallel algorithm on any parallel hardware” theme

Collaborations / External Studies

• early work at UIUC [1] [2]
• partnership with AMD [3] [4] [5]
• recent work from Georgia Tech and ANU, featured at CHIUW 2019 [6], CHIUW 2020 [7] and CHIUW 2021 [8]
• meanwhile, user applications have run on GPUs via Chapel interoperability features (e.g., ChOp and CHAMPS)

Rough Timeline

• **August 2020:** Design effort and discussions start
• **1.24 (March 2021):** Can use non-user-facing features to generate GPU binaries for Chapel functions and launch them
• **1.25 (September 2021):** Can natively translate order-independent loops into GPU kernels that are automatically launched
WHAT’S TO COME IN THIS TALK
GPU Codegen from Chapel

User’s loop
forall i in 1..n do arr[i] = i*mul;

The loop is replaced with:

if executingOnGPUSublocale()
    launch_kernel('kernel', n-1, 512, 1, 0,
                   n, 0, &arr, 32, mul, 0);
else
    for (i=1 ; i<=n ; i++) {
        int *arrData = arr->data;
        int *addrToChange = &arrData[i];
        int newVal = i*mul;
        *addrToChange = newVal;
    }

Generated GPU kernel looks like:

__global__
void kernel(int startIdx, int endIdx,
            int *arrArg, int mulArg) {
    int blockIdxX = __primitive('gpu blockIdx x');
    int blockDimX = __primitive('gpu blockDim x');
    int threadIdxX = __primitive('gpu threadIdx x');

    int t0 = blockIdxX * blockDimX;
    int t1 = t0 + threadIdxX;
    int index = t1 + startIdx;

    bool chpl_is_oob = index > endIdx;
    if (chpl_is_oob) { return; }

    int arrData = arrArg->data;
    int *addrToChange = &arrData[index];
    int newVal = myIdx*mulArg;
    *addrToChange = newVal;
}
PARALLELISM AND LOCALITY AS FIRST-CLASS CONCEPTS
THE LOCALE: CHAPEL’S KEY FEATURE FOR LOCALITY

- **locale**: a unit of the target architecture that can run tasks and store variables
  - Think “compute node” on a typical HPC system

---

**Locales array:**

- **Locale 0**
- **Locale 1**
- **Locale 2**
- **Locale 3**

**User’s program starts running as a single task on locale 0**

---

**Prompt:**

```c
prompt> ./myChapelProgram --numLocales=4  # or ‘-nl 4’
```
var x = 10;

on Locales[1] {
    var A = [1, 2, 3, 4, 5, ...];
    forall a in A do a += 1;
}

writeln(x);

Takeaway:
The ‘on’ statement and the ‘forall’ loop can be used to control locality and parallelism

Note: The ‘forall’ loop can result in distributed computation by itself, but that's out of scope for this talk
GPU PROGRAMMING AS FIRST-CLASS CONCEPTS
HIERARCHICAL LOCALES TO REPRESENT GPUS

GPUs represented as “sublocales”

Host cores and memory represented as normal

GPUs are just nested sublocales that have their own processing units and memory

Note: Specifics of the locale model design are an open discussion. See https://github.com/chapel-lang/chapel/issues/18529
PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUs

```
var x = 10;

on here.getGPU(0) {
    var A = [1, 2, 3, 4, 5, ...];
    forall a in A do a += 1;
}

writeln(x);
```
**OUR GOAL AND WHERE WE ARE**

Sample Computation: Status in 1.25

---

The ‘on’ statement moves the execution to a GPU sublocale

**Our Goal:**

```javascript
on here. getGPU(0) {
    var A = [1, 2, 3, 4, 5];
    forall a in A do a += 1;
}
```

Dynamic memory is allocated on the device

The ‘forall’ loop turns into a GPU kernel

**What works today (1.25.x):**

```javascript
on here. getChild(1) {
    var A = [1, 2, 3, 4, 5];
    forall a in A do a += 1;
}
```

Locale interface design is an ongoing work

---

**OUR GOAL AND WHERE WE ARE**

Sample Computation: Status in 1.25
Creating GPU Kernels from Loops

User’s loop
```c
forall i in 1..n do arr[i] = i*mul
```

Conceptual C loop
```c
for (i=1 ; i<=n ; i++) {
    // order-independent loop
    int *arrData = arr->data;
    int *addrToChange = &arrData[i];
    int newVal = i*mul;
    *addrToChange = newVal;
}
```

Loop body is copied

Generated GPU Kernel
```c
__global__
void kernel(int startIdx, int endIdx, int *arrArg, int mulArg) {
    int index = ...;  // calculate and return if >length
    int *arrData = arrArg->data;
    int *addrToChange = &arrData[index];
    int newVal = index*mulArg;
    *addrToChange = newVal;
}
```

The loop’s start and end indices are passed by value

Outer variables are passed depending on their type

Variables declared inside remain untouched

Symbols are replaced appropriately
Launching GPU Kernels

User’s loop
forall i in 1..n do arr[i] = i*mul

for (i=1 ; i<=n ; i++) {
    int *arrData = arr->data;
    int *addrToChange = &arrData[i];
    int newVal = i*mul;
    *addrToChange = newVal;
}

if executingOnGPUSublocale()
    launch_kernel('kernel', n-1, 512, 1, 0, n, 0, &arr, 32, mul, 0);
else
     // loop with no change

Kernel signature
__global__
void kernel(int startIdx, int length, int *arrArg, int mulArg);

Function name

Loop length and block size are used for dimension calculation

Pass-by-value arguments have an accompanying 0

Pass-by-offload arguments have an accompanying copy size

A dynamic check for GPU execution is added

Conceptual C loop

Generated Kernel Launch

User’s loop
forall i in 1..n do arr[i] = i*mul

for (i=1 ; i<=n ; i++) {
    int *arrData = arr->data;
    int *addrToChange = &arrData[i];
    int newVal = i*mul;
    *addrToChange = newVal;
}

if executingOnGPUSublocale()
    launch_kernel('kernel', n-1, 512, 1, 0, n, 0, &arr, 32, mul, 0);
else
     // loop with no change
**GPU CODEGEN: PART 3**

Translating Loop Indices Into Kernel Indices

---

**Kernel function**

```c
__global__
void kernel(int startIdx, int endIdx,
            int *arrArg, int mulArg) {

    int blockIdxX = __primitive('gpu blockIdx x');
    int blockDimX = __primitive('gpu blockDim x');
    int threadIdxX = __primitive('gpu threadIdx x');

    int t0 = blockIdxX * blockDimX;
    int t1 = t0 + threadIdxX;
    int index = t1 + startIdx;

    bool chpl_is_oob = index > endIdx;
    if (chpl_is_oob) { return; }

    // copied loop body
}
```

**Primitives correspond to CUDA threadIdx, blockIdx, blockDim, and gridDim variables**

They lower to calls to corresponding llvm intrinsics (e.g., `llvm.nvvm.read.ptx.sreg.ctaid.x`)

**Index computation**

Currently we are only targeting 1-dimensional kernels

**Check that index is in bounds**

Can occur if length is not evenly divisible by block size

**Loop body is copied**
**GPU CODEGEN**

**Putting the Pieces Together**

**User's loop**

```plaintext
forall i in 1..n do arr[i] = i*mul;
```

The loop is replaced with:

```plaintext
if executingOnGPUSublocale()
    launch_kernel("kernel", n-1, 512, 1, 0, n, 0, &arr, 32, mul, 0);
else
    for (i=1 ; i<=n ; i++) {
        int *arrData = arr->data;
        int *addrToChange = &arrData[i];
        int newVal = i*mul;
        *addrToChange = newVal;
    }
```

**Generated GPU kernel looks like:**

```c
__global__
void kernel(int startIdx, int endIdx,
            int *arrArg, int mulArg) {
    int blockIdxX = __primitive('gpu blockIdx x');
    int blockDimX = __primitive('gpu blockDim x');
    int threadIdxX = __primitive('gpu threadIdx x');

    int t0 = blockIdxX * blockDimX;
    int t1 = t0 + threadIdxX;
    int index = t1 + startIdx;

    bool chpl_is_oob = index > endIdx;
    if (chpl_is_oob) { return; }

    int arrData = arrArg->data;
    int *addrToChange = &arrData[index];
    int newVal = myIdx*mulArg;
    *addrToChange = newVal;
}
```
A Very Early Performance Study

Observations

- Can perform comparably to hand-written code
- Gets close to 100% efficiency with large datasets
- ‘foreach’ is slightly faster than ‘forall’

Potential Sources of Overhead

- Unified memory vs. device memory
- Dynamic allocations per kernel launch

Future Work for Performance

- Understand the performance with small vectors
- Profile the remaining costs
- Study other benchmarks
Summary
- Chapel’s language constructs for parallelism and locality suit GPU programming well
- The most recent Chapel release has a prototype feature for native GPU programming
- We:
  - have taken big steps in the recent releases
  - obtained very promising results both in terms of productivity and performance

Future Work
- A new locale model design
- Portability improvements
- Ability to create distributed arrays on GPUs
- Support more of the language features for GPU operations

GPU PROGRAMMING IN CHAPEL
CHAPEL RESOURCES

Chapel homepage: https://chapel-lang.org

Social Media:
- Twitter: @ChapelLanguage
- Facebook: @ChapelLanguage
- YouTube: http://www.youtube.com/c/ChapelParallelProgrammingLanguage

Community Discussion / Support:
- Discourse: https://chapel.discourse.group/
- Gitter: https://gitter.im/chapel-lang/chapel
- Stack Overflow: https://stackoverflow.com/questions/tagged/chapel
- GitHub Issues: https://github.com/chapel-lang/chapel/issues

GPU Technote:
- See https://chapel-lang.org/docs/master/technotes/gpu.html
THANK YOU

engin@hpe.com
linkedin.com/in/engink

chapel-lang.org