Declarative Parallel Programming for GPUs

Eric HOLK, William BYRD, Nilesh MAHAJAN, Jeremiah WILLCOCK, Arun CHAUHAN, Andrew LUMSDAINE

Indiana University, Bloomington, USA

ParCo 2011
September 1, 2011
Parallelism

Mainstream Parallelism-Oblivious Developers

Joe needs high level Programming Models designed for Domain Experts
Stephanie needs simple Parallel Programming Models with safety nets
Focus of today’s Parallel Programming Models

Concurrency Experts

Parallelism-Aware Developers

(Stephanie)

Joe

(Joe)

Arun Chauhan, Declarative parallel programming for GPUs, ParCo 2011

Courtesy: Vivek Sarkar, Rice University
Parallelism
Parallelism
Exa-scale Challenge
Design Principles

- Users must think in parallel (creativity)
  - but not be encumbered with optimizations that can be automated, or proving synchronization correctness

- Compiler focuses on what it can do (mechanics)
  - not creative tasks, such as determining data distributions, or creating new parallel algorithms

- Incremental deployment
  - not a new programming language
  - more of a coordination language (DSL)

- Formal semantics
  - provable correctness
Overview of Our Solution

- Declarative approach to parallel programming
  - focus on what, not how
- partitioned address space
- Code generation
  - data movement
  - GPU kernel splitting
- Compiler optimizations
  - data locality
  - GPU memory hierarchy (including registers)
Declarative Approach

- Originally motivated by Block-synchronous Parallel (BSP) programs, especially for collective communication
  - alternate between computation and communication
  - communication optimization breaks the structure
Declarative Approach

- Originally motivated by Block-synchronous Parallel (BSP) programs, especially for collective communication
- alternate between computation and communication
- communication optimization breaks the structure
Declarative Approach

- Originally motivated by Block-synchronous Parallel (BSP) programs, especially for collective communication
  - alternate between computation and communication
  - communication optimization breaks the structure
Declarative Approach

• Originally motivated by Block-synchronous Parallel (BSP) programs, especially for collective communication
  • alternate between computation and communication
  • communication optimization breaks the structure

• Extend to non BSP-style applications
Kanor for Clusters

@communicate { b@recv_rank <= a@send_rank }

Kanor for Clusters

@communicate { b@recv_rank <= a@send_rank }

e_0 @ e_1 << op << e_2 @ e_3 where e_4
Kanor for Clusters

@communicate { b@recv_rank <= a@send_rank }

e_0 @ e_1 << op << e_2 @ e_3 where e_4

e_0 @ e_1 <<= e_2 @ e_3 where e_4
Kanor for Clusters

@communicate { b@recv_rank <= a@send_rank }
Kanor for Clusters

@communicate { b@recv_rank <= a@send_rank }

e_0 @ e_1 << op << e_2 @ e_3 where e_4

e_0 @ e_1 <= e_2 @ e_3 where e_4

A[j] @ i <= B[i] @ j where i in world, j in {0...i}, i % 2 == 0

storage location receiver rank reduction operator data sender rank generator generator filter

Source-level compiler (using ROSE)

standard C++ code


Arun Chauhan, Declarative parallel programming for GPUs, ParCo 2011
Distributed Memory Targets

- Generate MPI
- Recognize collectives that map to MPI collectives
- Optimize communication
  - computation-communication overlap
  - communication coalescing
Shared Memory Targets

- Use partitioned address space
- Leverage shared memory for communication
- Eliminate buffer copying
  - identify opportunities for aliasing
  - insert synchronization for correctness
  - optimize at run time to eliminate synchronization overheads

@communicate\{x@i <= x@0\}, where i > 0

... = x

x = ...

x = ...

... = x

x = ...

... = x

x = ...

... = x

x = ...

... = x

E


Arun Chauhan, Declarative parallel programming for GPUs, ParCo 2011
Harlan for GPUs

```c
__global__ void add_kernel(int size, float *X, float *Y, float *Z) {
    int i = threadIdx.x;
    if (i < size) { Z[i] = X[i] + Y[i]; }
}

void vector_add(int size, float *X, float *Y, float *Z) {
    float *dX, *dY, *dZ;
    cudaMalloc(&dX, size * sizeof(float));
    cudaMalloc(&dY, size * sizeof(float));
    cudaMalloc(&dZ, size * sizeof(float));

    cudaMemcpy(dX, X, size * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dY, Y, size * sizeof(float), cudaMemcpyHostToDevice);

    add_kernel <<<1, size >>>(size, dX, dY, dZ);
    cudaMemcpy(Z, dZ, size * sizeof(float), cudaMemcpyDeviceToHost);

    cudaFree(dX);
    cudaFree(dY);
    cudaFree(dZ);
}
```

Figure 1. CUDA code for adding two vectors.

```c
void vector_add(vector<float>X, vector<float>Y, vector<float>Z) {
    kernel(x : X, y : Y, z : Z) { z = x + y; }
}
```

Figure 2. Harlan code for adding two vectors.

Arun Chauhan, Declarative parallel programming for GPUs, ParCo 2011
Harlan for GPUs

void vector_add (vector<float> X, vector<float> Y, vector<float> Z) {
    kernel (x : X, y : Y, z : Z) { z = x + y; }
}
Harlan Features

Reductions

\[ z = +/\text{kernel} \ (x : X, \ y : Y) \{ \ x * y \}; \]
Harlan Features

Reductions

\[ z = \text{+/kernel} (x : X, y : Y) \{ x \times y \}; \]

Asynchronous kernels

handle = \text{async kernel} (x : X, y : Y) \{ x \times y \};

// other concurrent kernels of program code here
z = \text{+/wait}(handle);
Harlan Features

Reductions

\[
z = +/\text{kernel} \ (x : X, y : Y) \{ \ x \ast y \ \};
\]

Asynchronous kernels

\[
\text{handle} = \text{async \ kernel} \ (x : X, y : Y) \{ \ x \ast y \ \};
// \ other \ concurrent \ kernels \ of \ program \ code \ here
z = +/\text{wait}(\text{handle});
\]

Nested kernels

\[
\text{total} = +/\text{kernel} \ (\text{row} : \text{Rows}) \{ +/\text{kernel} \ (x : \text{row}); \ \};
\]
Example 1: Dot Product

```cpp
// dot product of two vectors
double dotproduct(Vector X, Vector Y) {
    double dot = +/-kernel(x : X, y : Y) { x * y };
    return dot;
}
```
Example 2: Dense Matrix Multiply

// dense matrix-matrix multiply
Matrix matmul (Matrix A, Matrix B) {
  // this block does a transpose; it could go in a library
  Bt = kernel(j : [0 .. length(B[0])]) {
    kernel(i : [0 .. length(B)]) {
      B[j][i];
    }
  };
  C = kernel(row : A) {
    kernel(col : Bt) {
      +/-kernel(a : row, b : col) {
        a * b;
      }
    }
  }
  return C;
}
Example 3: Sparse Mat-Vec Product

// sparse matrix-vector product (CSR)
Vector spmv(CSR_i Ai, CSR_v Av, Vector X) {
  Vector Y = kernel(is : Ai, vs : Av) {
    |/kernel(i : is, v : vs) { v * X[i]; }
  }
  return Y;
}
Combining Kanor and Harlan

```plaintext
kernel (x : X, y : Y, z : Z) {
  z = x * y;
}

@communicate {
  Y[i]@r <= Z[i]@((r+1) & NUM_NODES)
  where r in world,
    i in 0...length(Y)
}

kernel (x : X, y : Y, z : Z) {
  z = x * y;
}
```
Code Generation

- Data transfers between CPU and device memory
  - hide or minimize data movement latency
- Kernel splitting
  - to accommodate the limitations of GPUs
Optimizations

• Data movement
  • account for data locality
  • only move live data needed

• Kernel splitting
  • smaller kernels might increase concurrency

• Scheduling concurrent kernels

• Scheduling reduction

• Mapping variables within GPU memory hierarchy

• Optimizing thread count
Experiments

Platform:

2.8 GHz Quad-Core Intel Xeon
8GB 1066 MHz DDR3 RAM
ATI Radeon HD 5770 1024MB
Mac OS X Lion 10.7.1
Vector Dot Product
Concluding Remarks

• Declarative approach to parallelism
  • focus on what, now how
  • divide the work between user and software according to their strengths

• Variety of parallel platforms
  • Kanor: declarative parallelism for clusters
  • Harlan: declarative parallelism for GPUs
  • Combination: declarative parallelism for GPU clusters

• Optimizations through a combination of compiler analysis, smart run time system, and auto-tuning
Questions?
Neighbor Communication

```plaintext
kernel(x : X, y : Y) {
    y = 0.25 * (x.east + x.west + x.north + x.south);
}
```