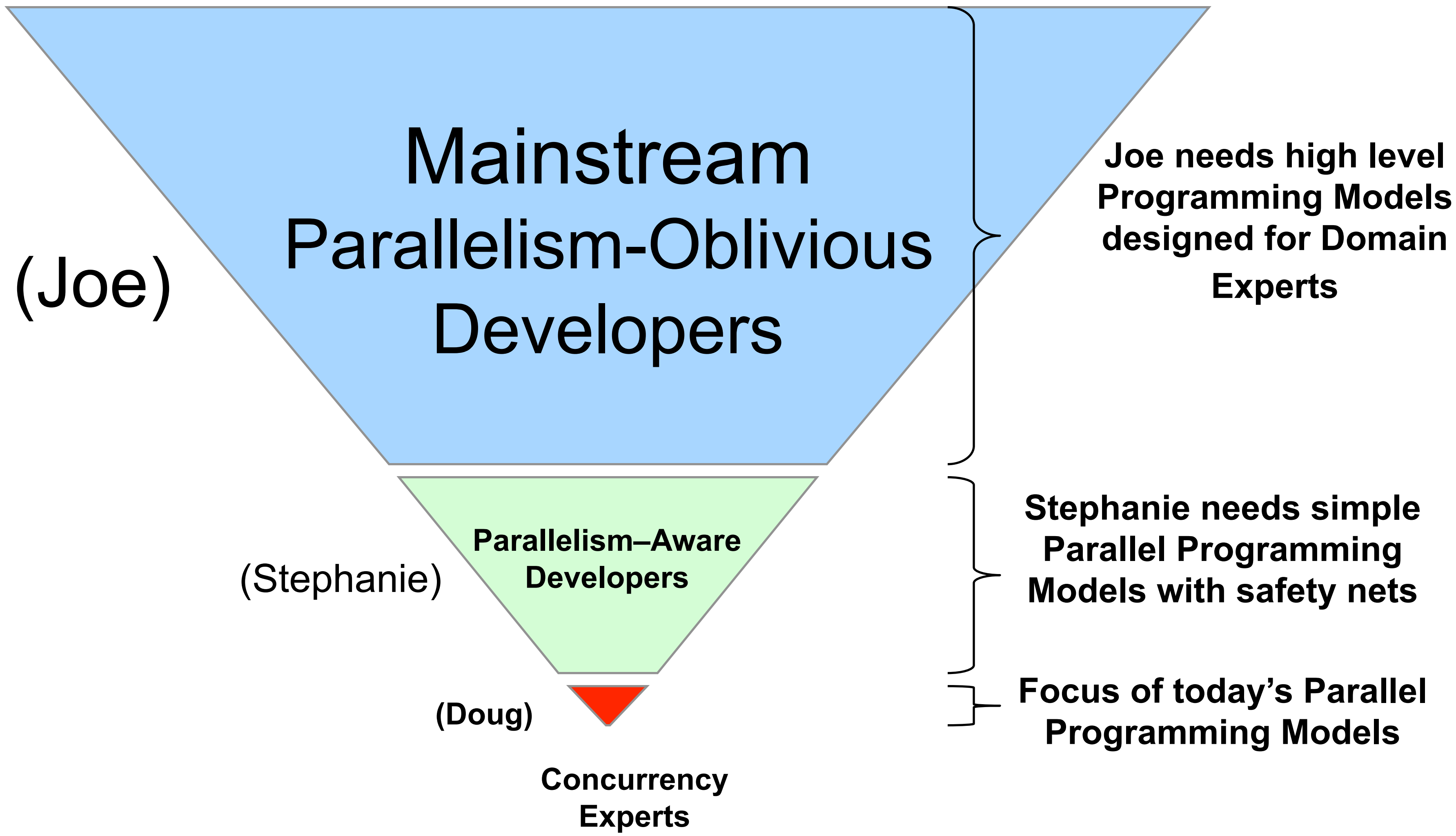
Declarative Parallel Programming for GPUs

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)

Joe needs high level Programming Models designed for Domain Experts

Parallelism–Aware Developers

(Stephanie)

Stephanie needs simple Parallel Programming Models with safety nets

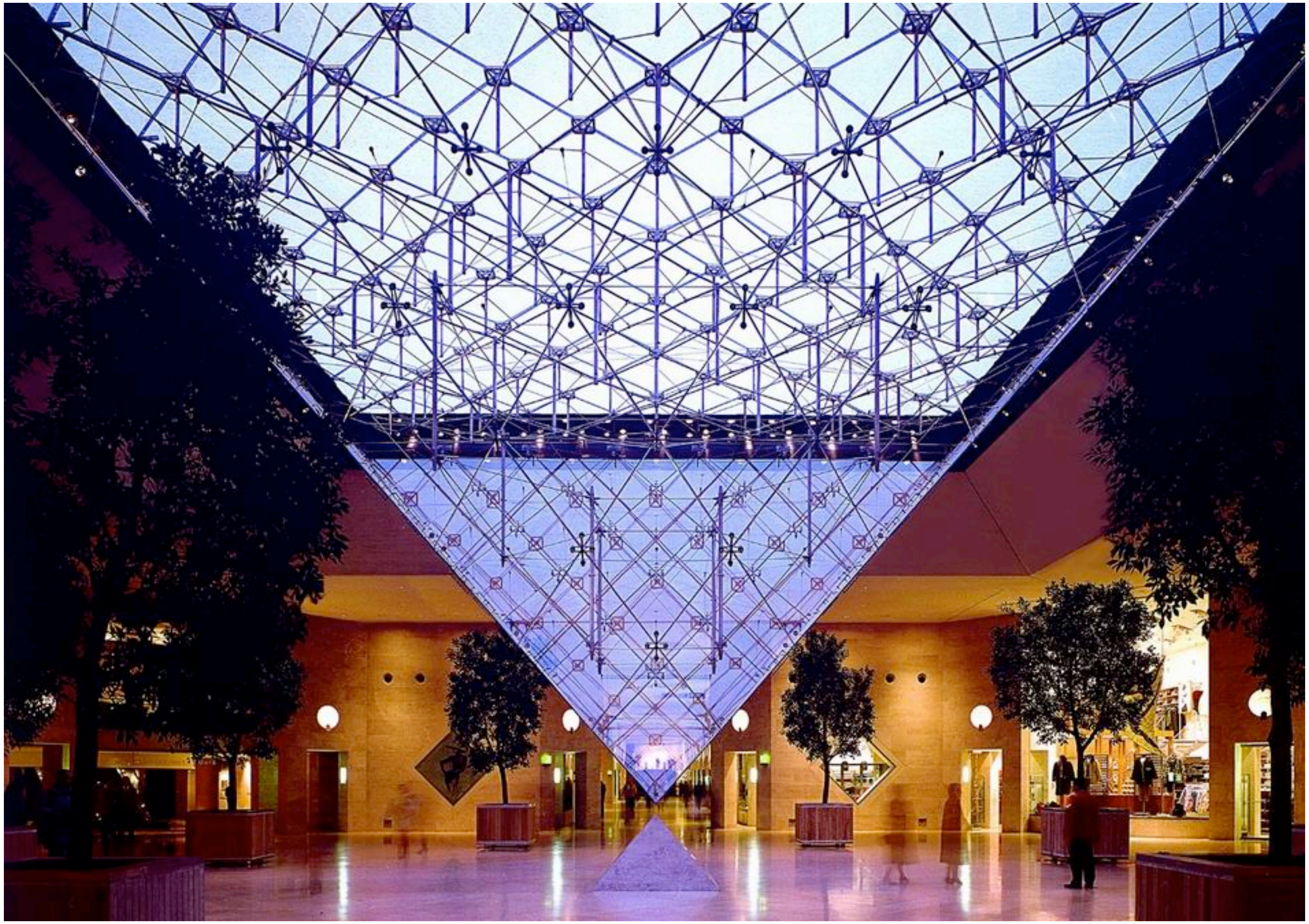
(Doug)

Concurrency Experts

Focus of today's Parallel Programming Models

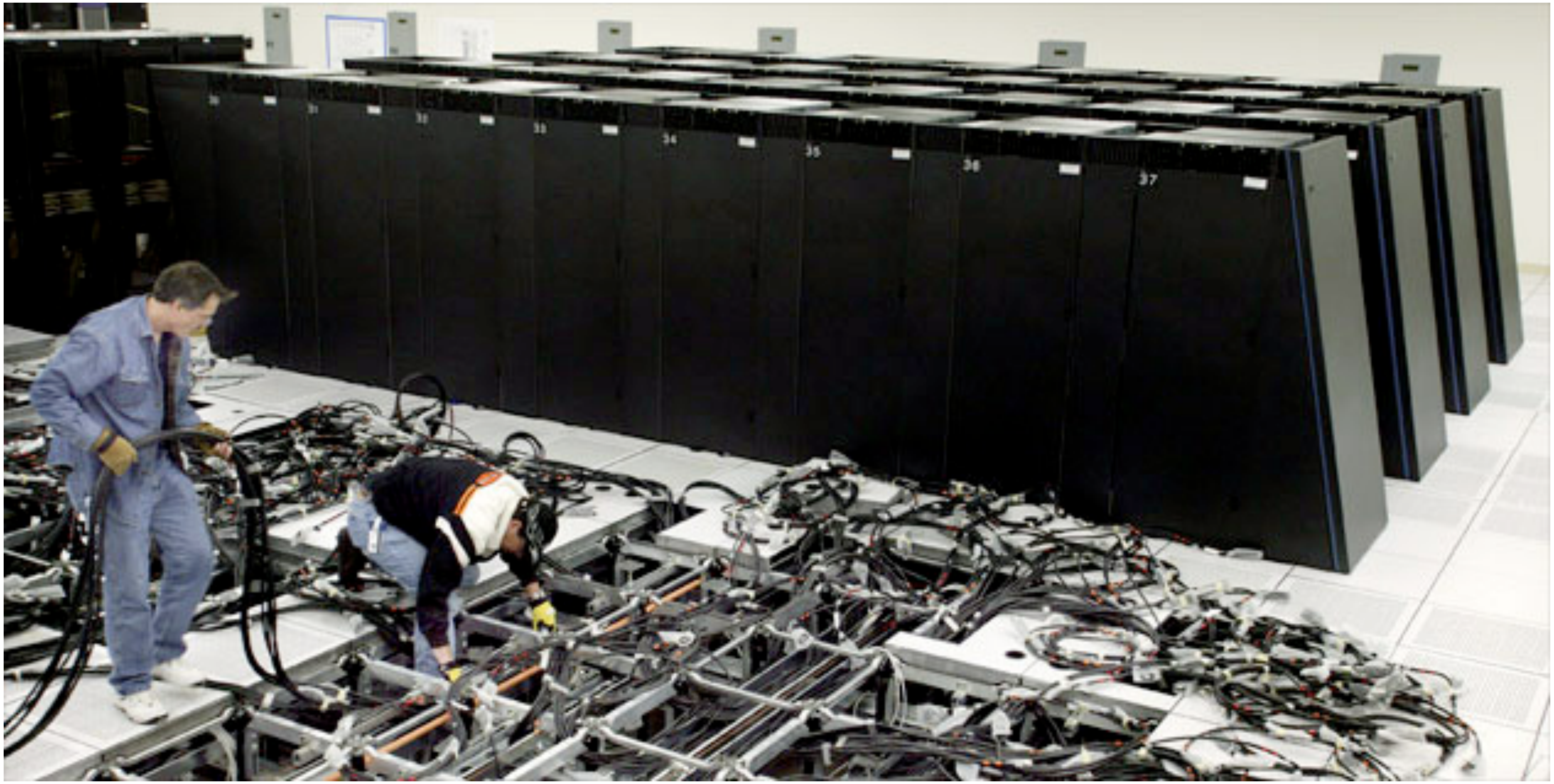
Courtesy: Vivek Sarkar, Rice University

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)

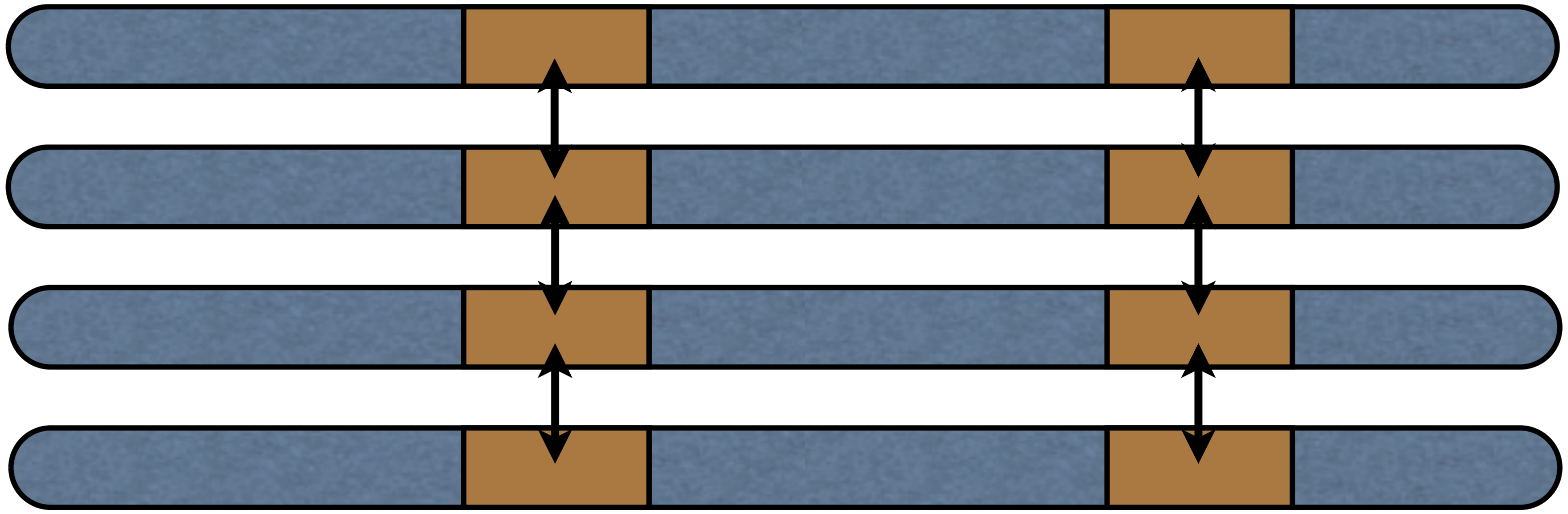
Torsten Hoefler, Jeremiah Willcock, Arun Chauhan, and Andrew Lumsdaine. **The Case for Collective Pattern Specification**. In *Proceedings of the First Workshop on Advances in Message Passing (AMP)*, 2010. Held in conjunction with the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI).

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

$$\textbf{@communicate}\ \{\ b@recv\_rank <<= a@send\_rank\ \}$$

Eric Holk, William E. Byrd, Jeremiah Willcock, Torsten Hoefler, Arun Chauhan, and Andrew Lumsdaine. **Kanor: A Declarative Language for Explicit Communication**. In *Proceedings of the Thirteenth International Symposium on the Practical Aspects of Declarative Languages (PADL)*, 2011. Held in conjunction with the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL).

Kanor for Clusters

$$\mathbf{@communicate}\ \{\ b@recv\_rank <<= a@send\_rank\ \}$$

$$e_0@e_1 << op << e_2@e_3 \texttt{ where } e_4$$

Eric Holk, William E. Byrd, Jeremiah Willcock, Torsten Hoefler, Arun Chauhan, and Andrew Lumsdaine. **Kanor: A Declarative Language for Explicit Communication**. In *Proceedings of the Thirteenth International Symposium on the Practical Aspects of Declarative Languages (PADL)*, 2011. Held in conjunction with the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL).

Kanor for Clusters

$$\textbf{@communicate} \; \{ \; b@\mathit{recv\_rank} <<= a@\mathit{send\_rank} \; \}$$

$$e_0@e_1 << \mathit{op} << e_2@e_3 \; \texttt{where} \; e_4$$

$$e_0@e_1 <<= e_2@e_3 \; \texttt{where} \; e_4$$

Eric Holk, William E. Byrd, Jeremiah Willcock, Torsten Hoefler, Arun Chauhan, and Andrew Lumsdaine. **Kanor: A Declarative Language for Explicit Communication**. In *Proceedings of the Thirteenth International Symposium on the Practical Aspects of Declarative Languages (PADL)*, 2011. Held in conjunction with the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL).

Kanor for Clusters

$$\textbf{@communicate}\ \{\ b@recv\_rank \ {<}{<}{=}\ a@send\_rank\ \}$$

$$e_0@e_1 \ {<}{<}\ op\ {<}{<}\ e_2@e_3\ \texttt{where}\ e_4$$

$$e_0@e_1 \ {<}{<}{=}\ e_2@e_3\ \texttt{where}\ e_4$$

$$\underbrace{\texttt{A[j]}}_{\text{storage location}}\ @\ \underbrace{\texttt{i}}_{\text{receiver rank}}\ \underbrace{\texttt{<<=}}_{\text{reduction operator}}\ \underbrace{\texttt{B[i]}}_{\text{data}}\ @\ \underbrace{\texttt{j}}_{\text{sender rank}}\ \texttt{where}\ \underbrace{\texttt{i in world,}}_{\text{generator}}\ \underbrace{\texttt{j in \{0...i\},}}_{\text{generator}}\ \underbrace{\texttt{i \% 2 == 0}}_{\text{filter}}$$

Eric Holk, William E. Byrd, Jeremiah Willcock, Torsten Hoefler, Arun Chauhan, and Andrew Lumsdaine. **Kanor: A Declarative Language for Explicit Communication**. In *Proceedings of the Thirteenth International Symposium on the Practical Aspects of Declarative Languages (PADL)*, 2011. Held in conjunction with the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL).

Kanor for Clusters

$$\textbf{@communicate}\ \{\ b@recv\_rank <<= a@send\_rank\ \}$$

$$e_0 @ e_1 << op << e_2 @ e_3\ \texttt{where}\ e_4$$

$$e_0 @ e_1 <<= e_2 @ e_3\ \texttt{where}\ e_4$$

$$\underbrace{\texttt{A[j]}}_{\text{storage location}}\ @\ \underbrace{\texttt{i}}_{\text{receiver rank}}\ \underbrace{\texttt{<<=}}_{\text{reduction operator}}\ \underbrace{\texttt{B[i]}}_{\text{data}}\ @\ \underbrace{\texttt{j}}_{\text{sender rank}}\ \texttt{where}\ \underbrace{\texttt{i in world}}_{\text{generator}},\ \underbrace{\texttt{j in \{0...i\}}}_{\text{generator}},\ \underbrace{\texttt{i \% 2 == 0}}_{\text{filter}}$$

↓

Source-level compiler (using ROSE)

↓

standard C++ code

Eric Holk, William E. Byrd, Jeremiah Willcock, Torsten Hoefler, Arun Chauhan, and Andrew Lumsdaine. **Kanor: A Declarative Language for Explicit Communication**. In *Proceedings of the Thirteenth International Symposium on the Practical Aspects of Declarative Languages (PADL)*, 2011. Held in conjunction with the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL).

Distributed Memory Targets

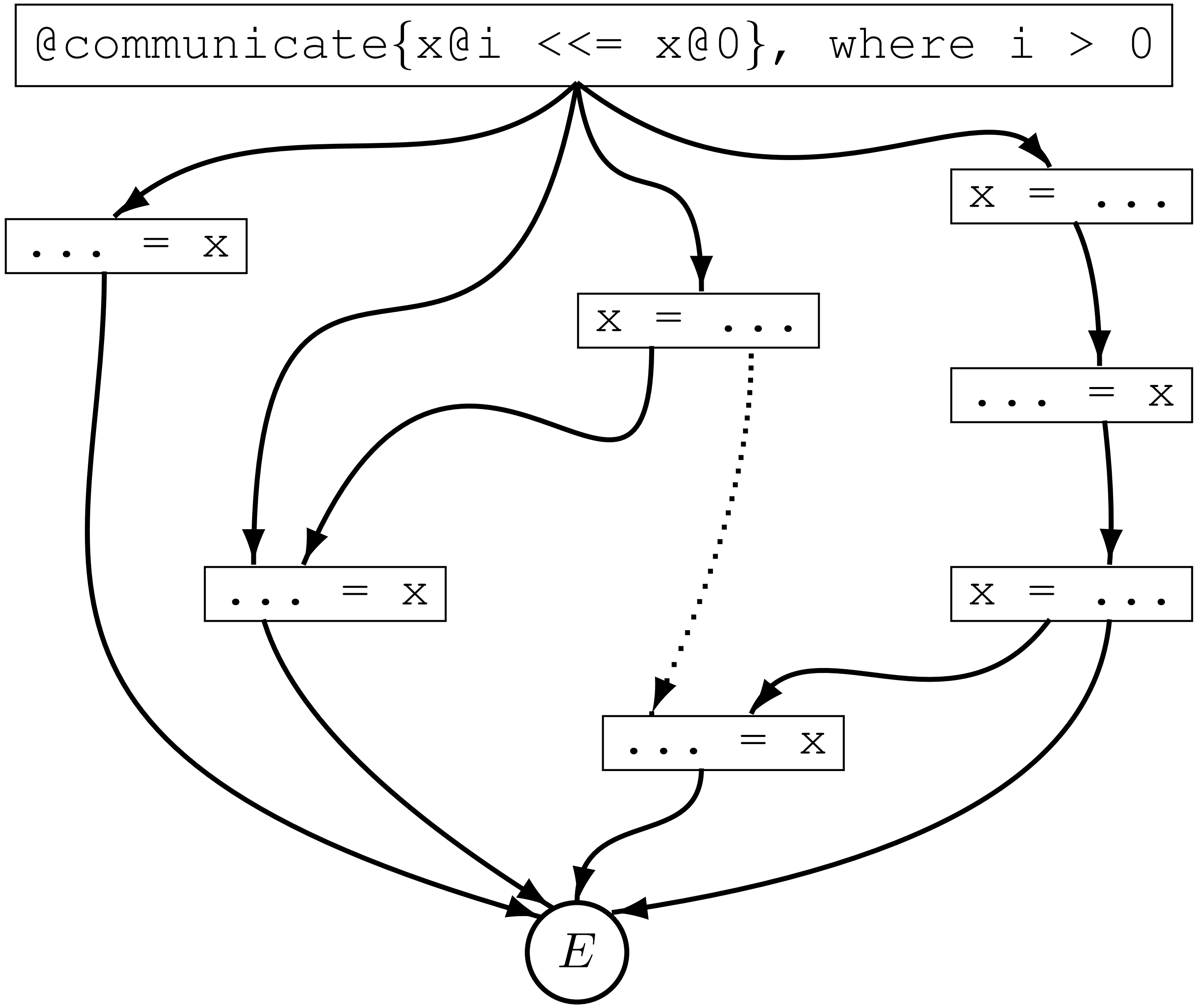
- Generate MPI
- Recognize collectives that map to MPI collectives
- Optimize communication
  - computation-communication overlap
  - communication coalescing

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

Fangzhou Jiao, Nilesh Mahajan, Jeremiah Willcock, Arun Chauhan, and Andrew Lumsdaine. **Partial Globalization of Partitioned Address Space for Zero-copy Communication with Shared Memory**. In *Proceedings of the 18th International Conference on High Performance Computing (HiPC)*, 2011. *To appear*.

Optimizing for Shared Memory

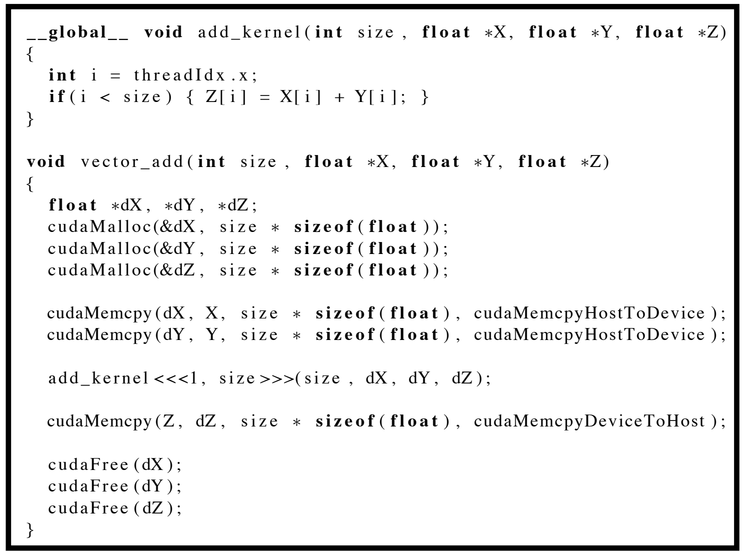


Fangzhou Jiao, Nilesh Mahajan, Jeremiah Willcock, Arun Chauhan, and Andrew Lumsdaine. **Partial Globalization of Partitioned Address Space for Zero-copy Communication with Shared Memory**. In *Proceedings of the 18th International Conference on High Performance Computing (HiPC)*, 2011. *To appear*.

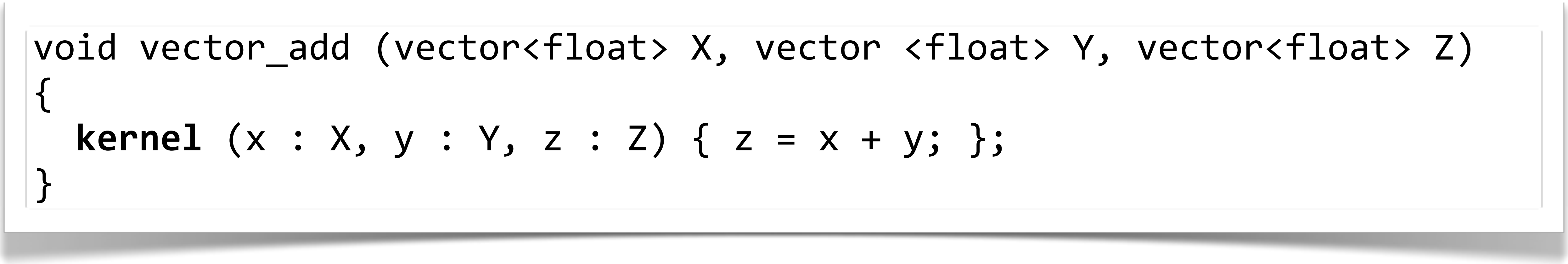
Harlan for GPUs

```
__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);
}
```

Harlan for GPUs



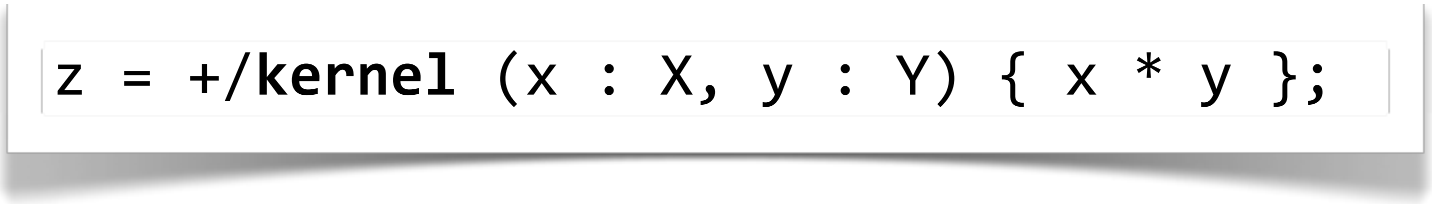
```
__global__ void add_kernel(int size, float *X, float *Y, float *Z)
{
  int i = threadIdx.x;
  if(i < size) { Z[i] = X[i] + Y[i]; }
}

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);
}
```



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

Harlan Features

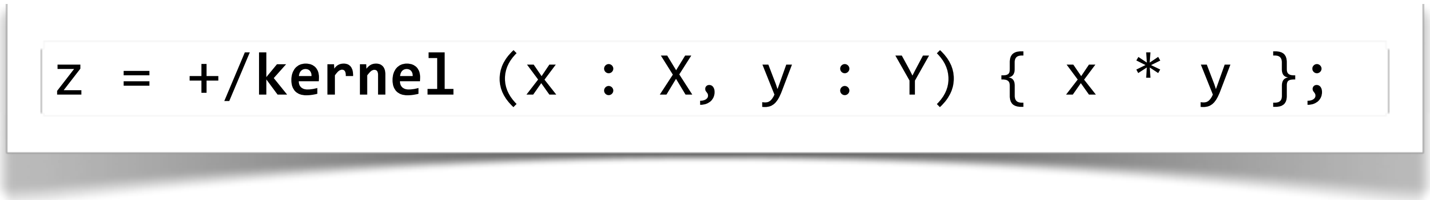
Reductions



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

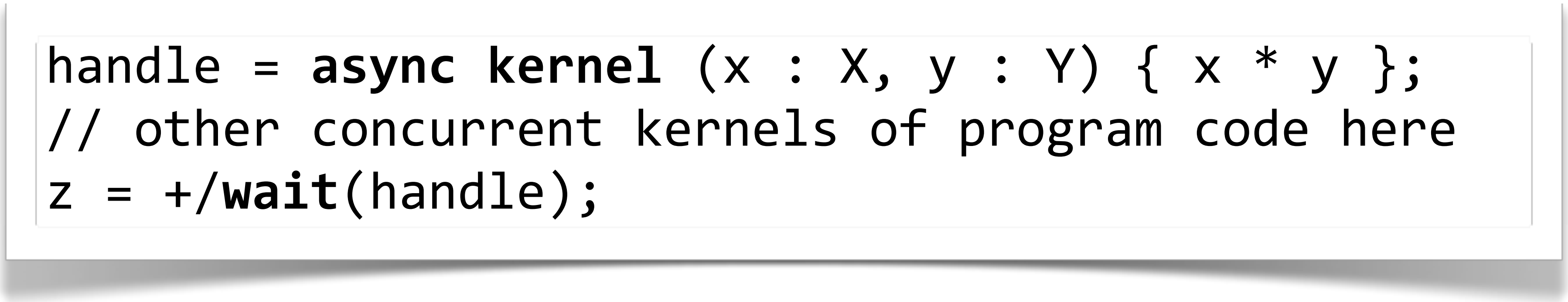
Harlan Features

Reductions



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

Asynchronous kernels



```
handle = async kernel (x : X, y : Y) { x * y };
// other concurrent kernels of program code here
z = +/wait(handle);
```

Harlan Features

Reductions

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

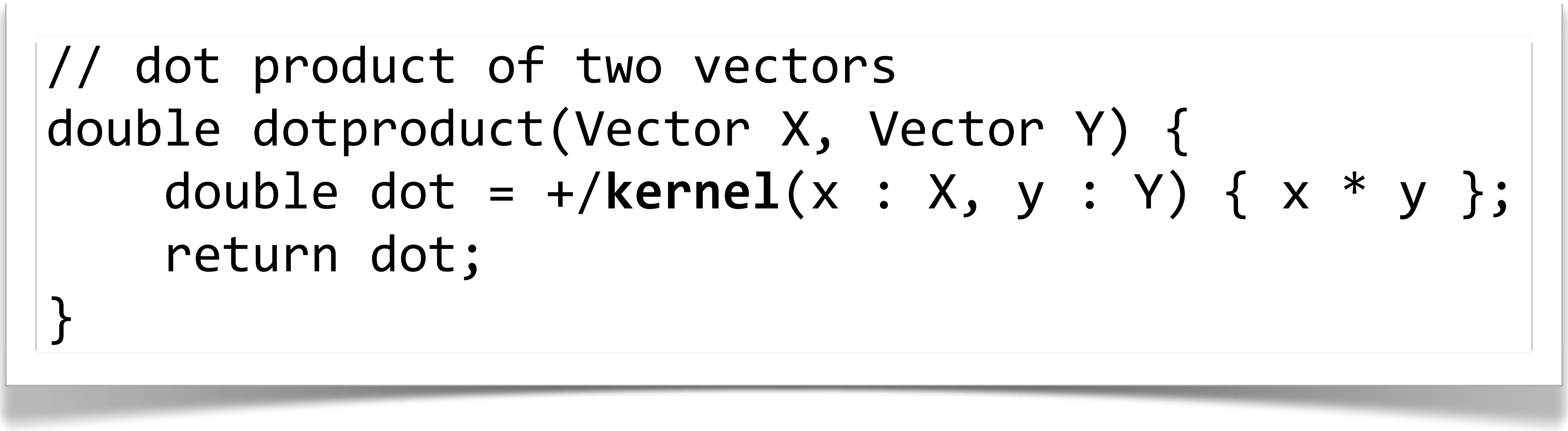
Asynchronous kernels

```
handle = async kernel (x : X, y : Y) { x * y };
// other concurrent kernels of program code here
z = +/wait(handle);
```

Nested kernels

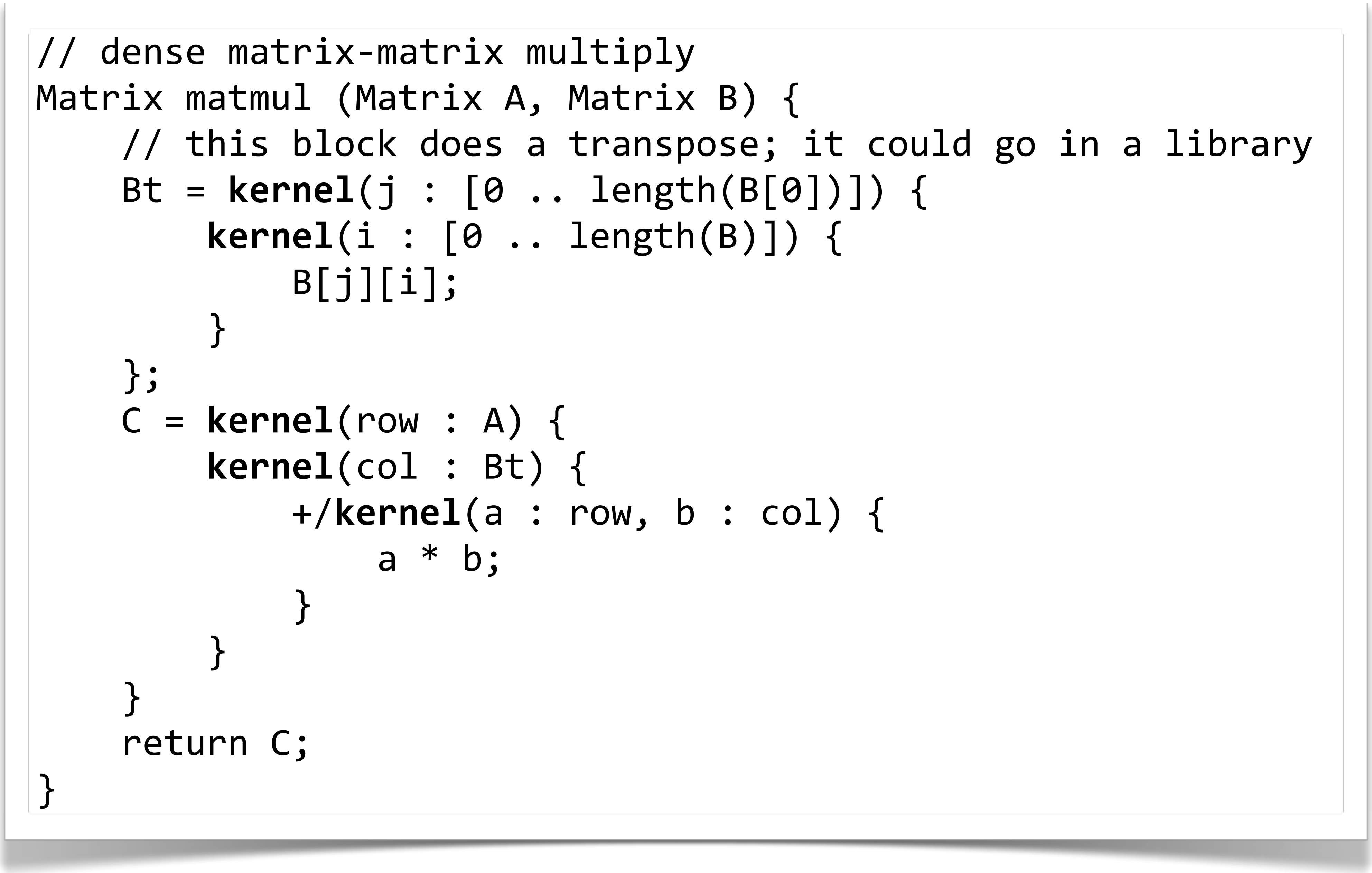
```
total = +/kernel (row : Rows) { +/kernel (x : row); };
```

Example 1: Dot Product



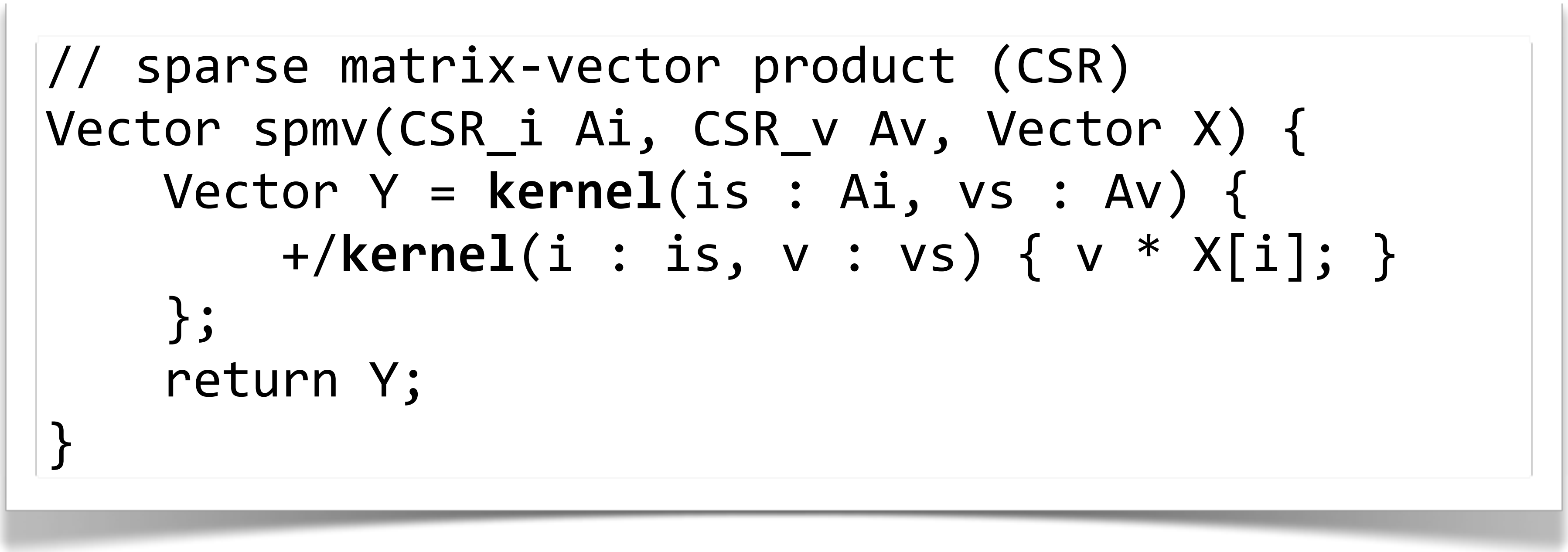
```
// dot product of two vectors
double dotproduct(Vector X, Vector Y) {
    double dot = +/kernel(x : X, y : Y) { x * y };
    return dot;
}
```

Example 2: Dense Matrix Multiply



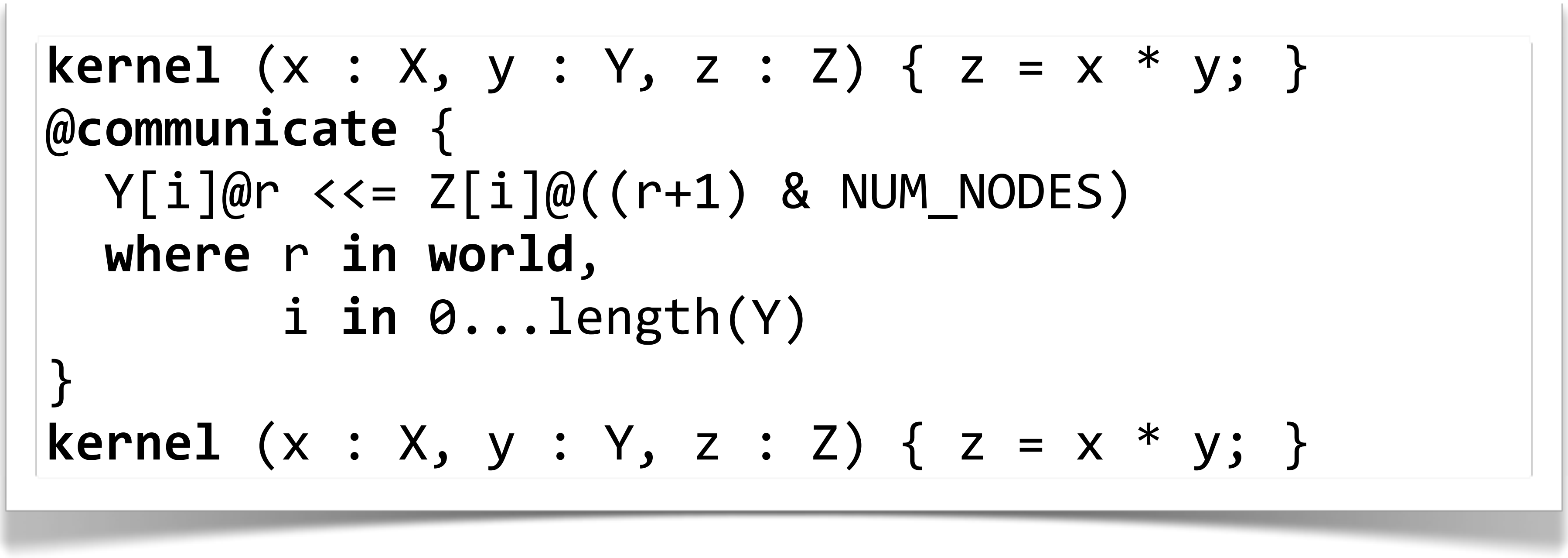
```
// dense matrix-matrix multiply
Matrix matmul (Matrix A, Matrix B) {
    // this block does a transpose; it could go in a library
    Bt = kernel(j : [0 .. length(B[0])]) {
        kernel(i : [0 .. length(B)]) {
            B[j][i];
        }
    };
    C = kernel(row : A) {
        kernel(col : 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



```
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

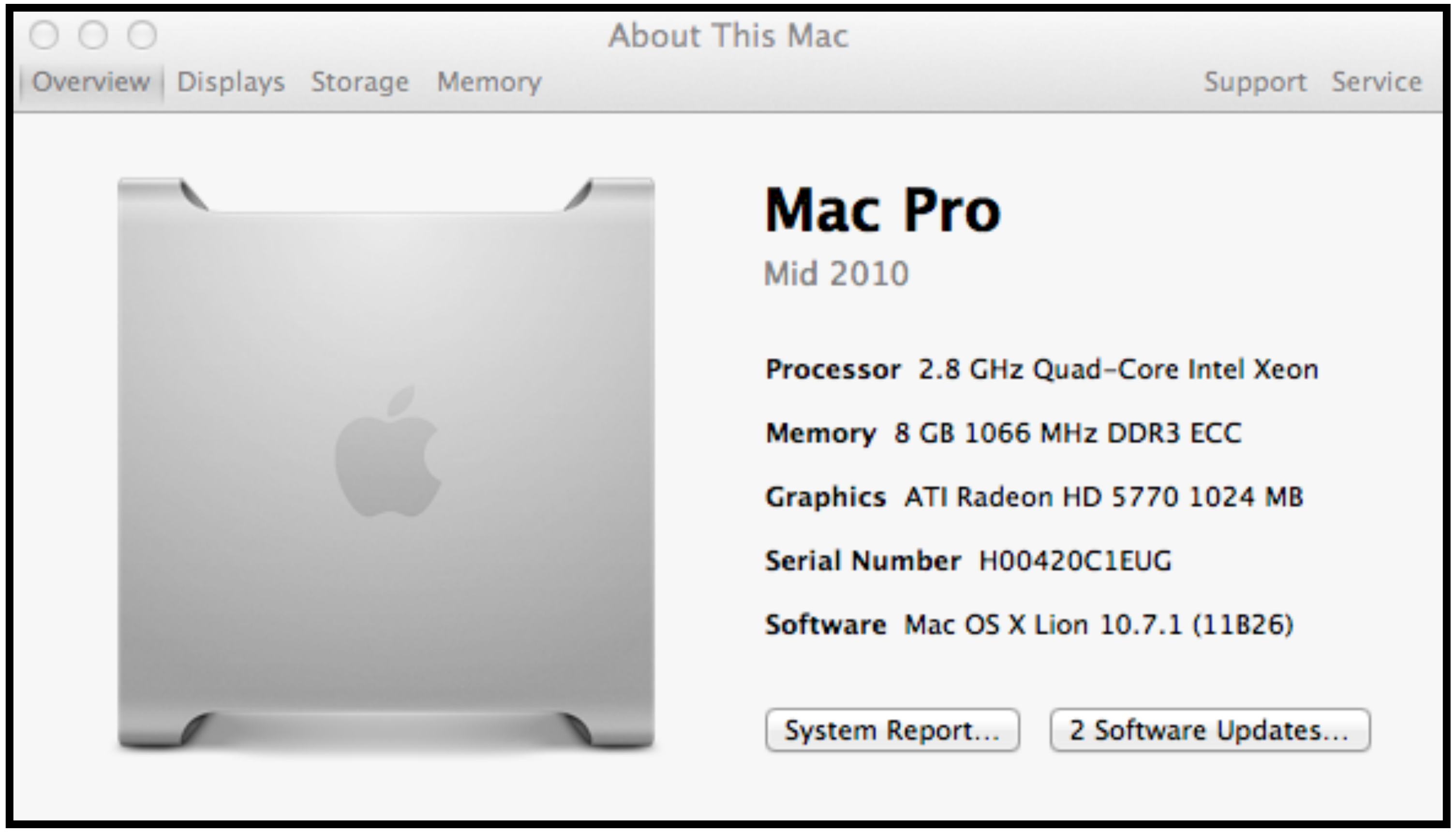
Platform:

2.8 GHz Quad-Core Intel Xeon

8GB 1066 MHz DDR3 RAM

ATI Radeon HD 5770 1024MB

Mac OS X Lion 10.7.1



Mac Pro

Mid 2010

Processor 2.8 GHz Quad-Core Intel Xeon

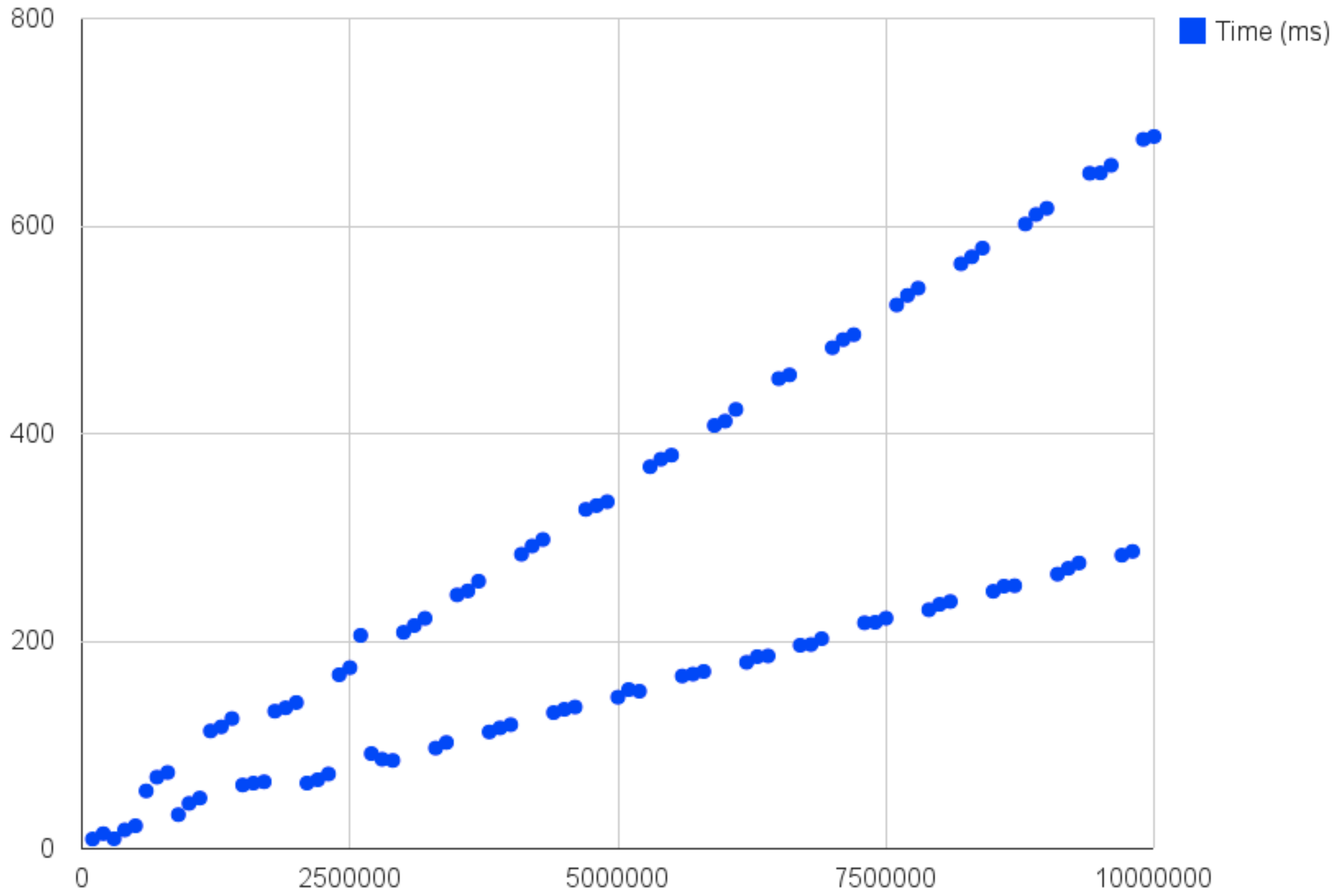
Memory 8 GB 1066 MHz DDR3 ECC

Graphics ATI Radeon HD 5770 1024 MB

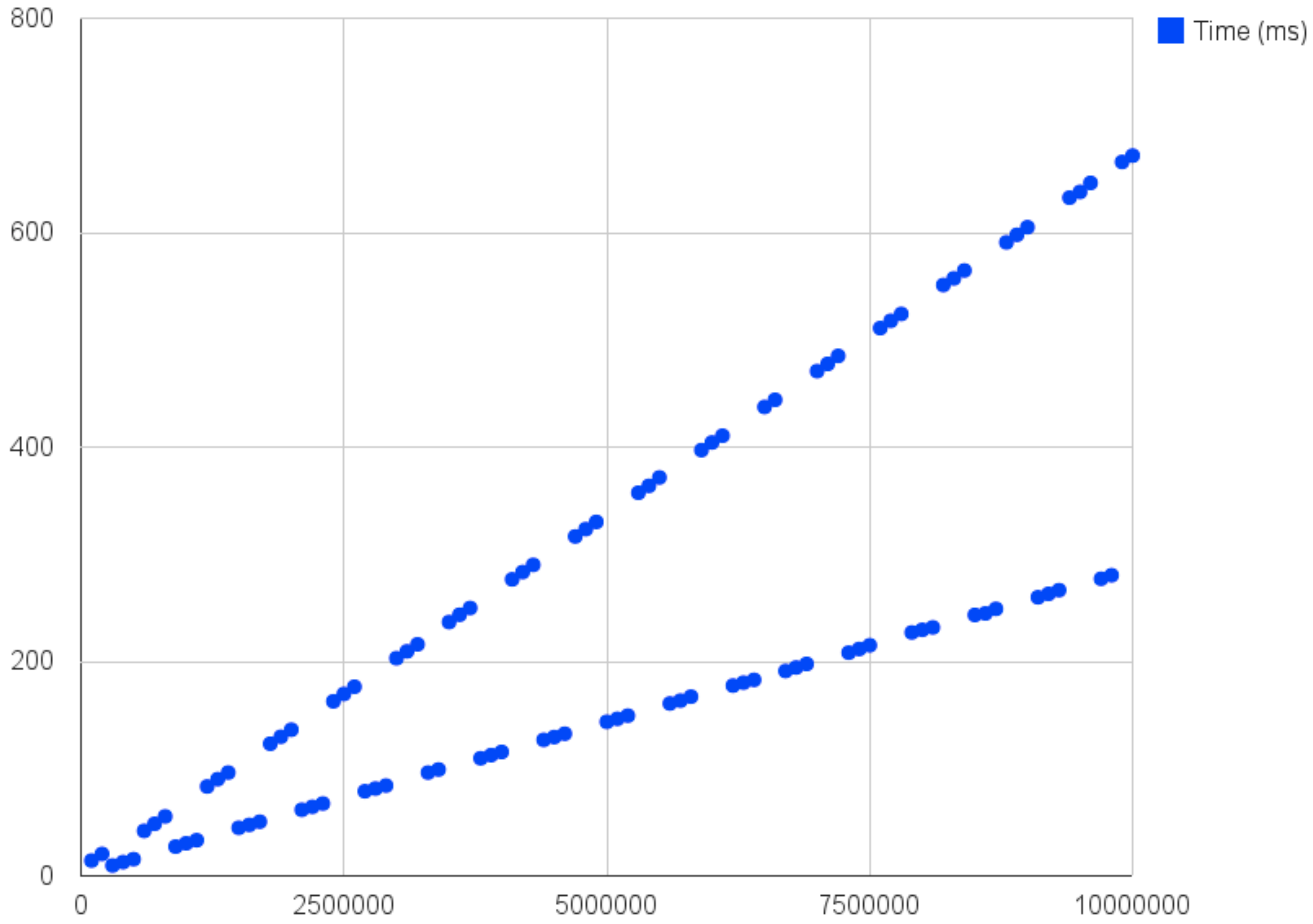
Serial Number H00420C1EUG

Software Mac OS X Lion 10.7.1 (11B26)

Vector Dot Product

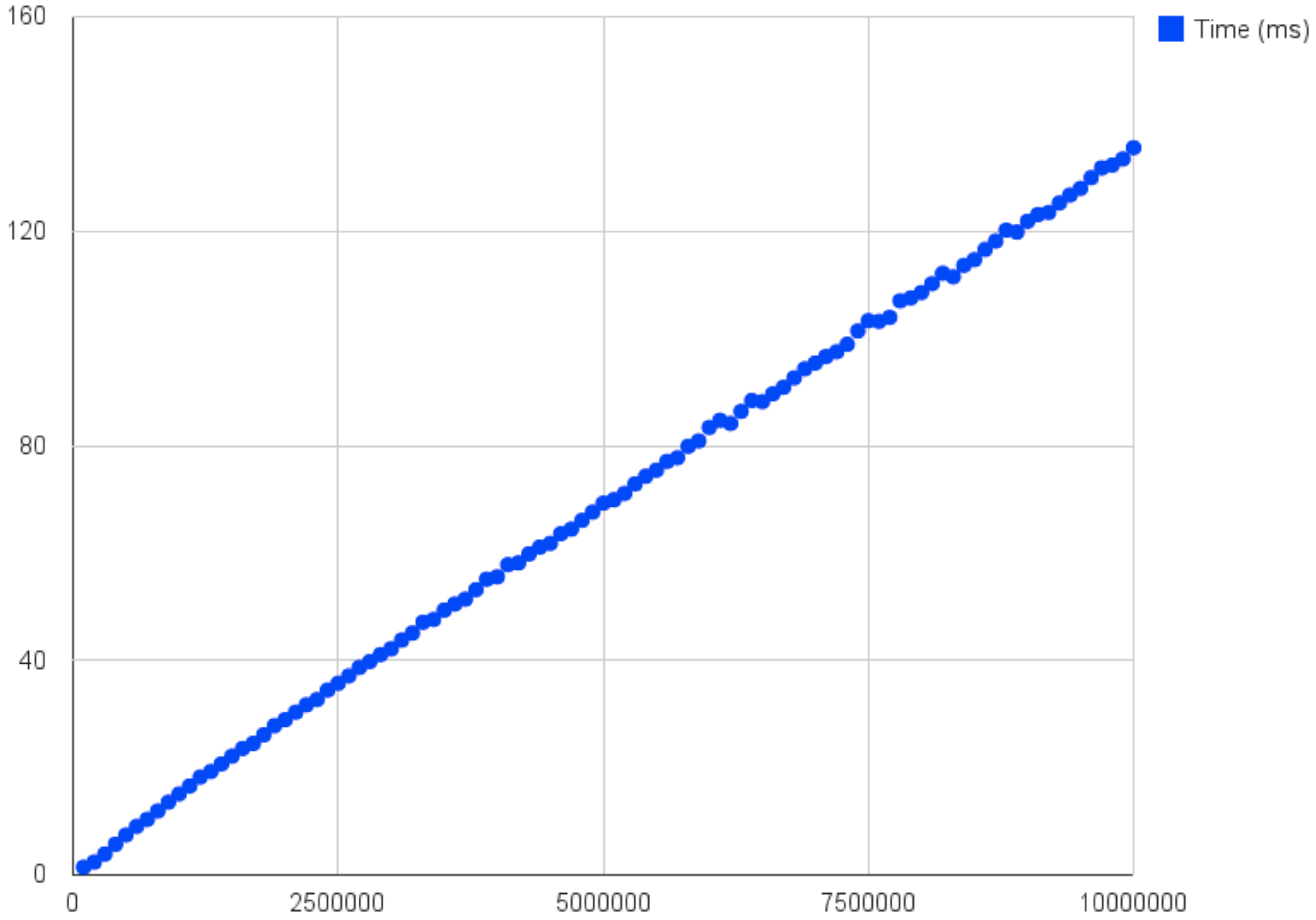


Vector Sum



Arun Chauhan, Declarative parallel programming for GPUs, ParCo 2011

Dot Product (CPU)

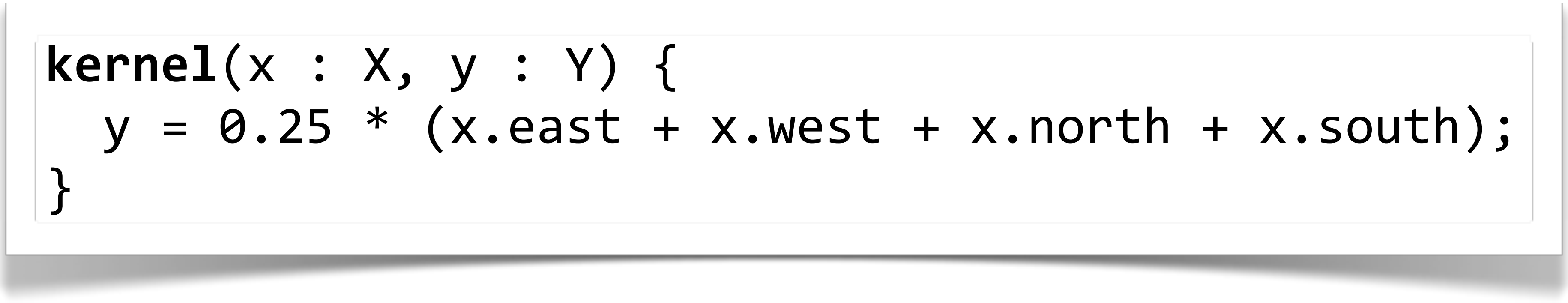


Concluding Remarks

- 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



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