# Performance Diagnosis for Hybrid CPU/GPU Environments

Michael M. Smith and Karen L. Karavanic

Computer Science Department

Portland State University

Portland State UNIVERSITY

# Performance Diagnosis for Hybrid CPU/GPU Environments

Michael M. Smith and Karen L. Karavanic

Computer Science Department

Portland State University

*Work in Progress*

Portland State UNIVERSITY

# Introduction

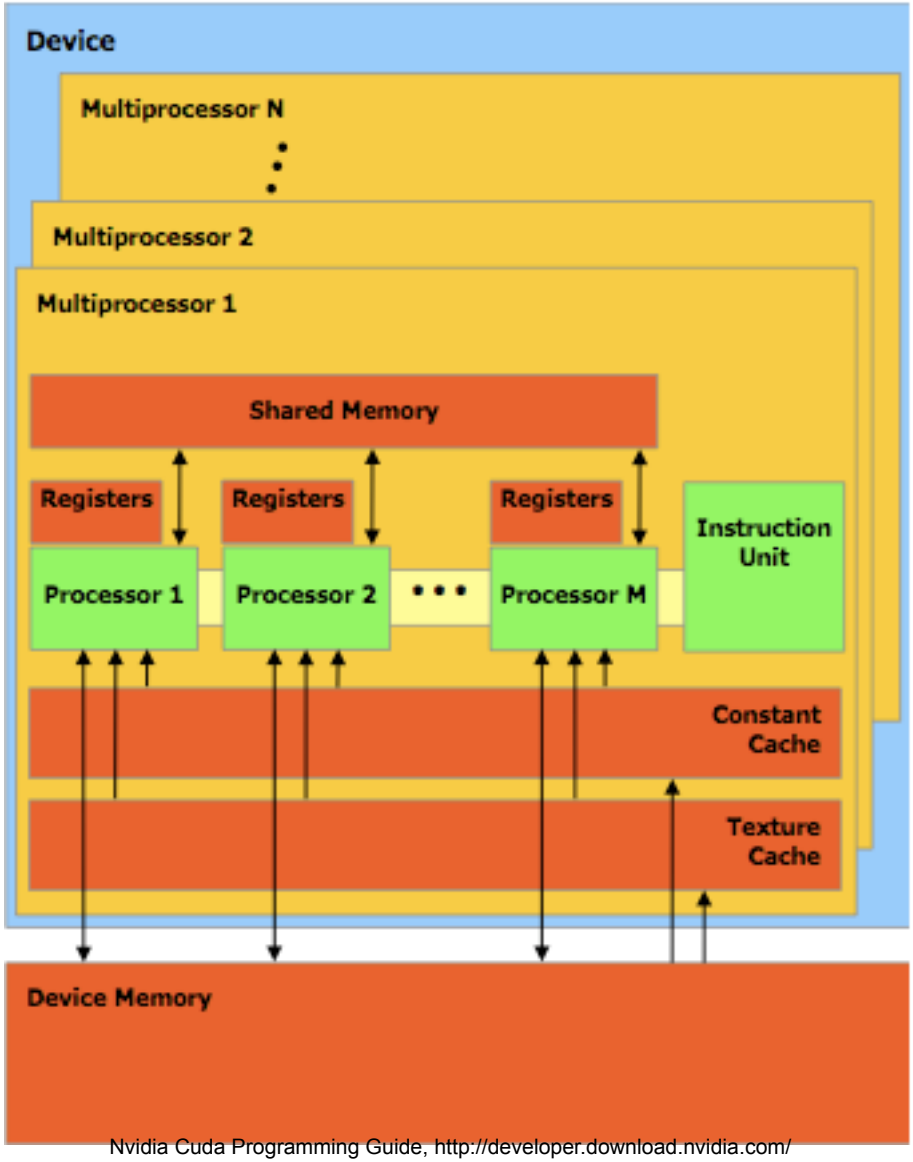
Shift to GPUs:

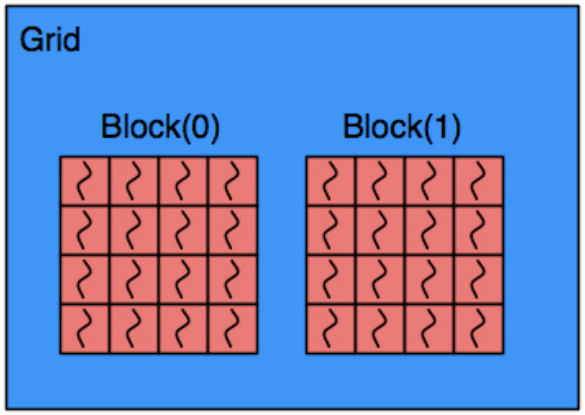
- Single core processors have hit performance wall due to heat dissipation and power requirements
- => multicore, manycore
- Top500 (June 2011):
  - 17/500 are GPU-accelerated
  - Includes #2 (Tianhe-1a), #4 (Nebulae) and #5 (Tsubame 2.0)
- Hybrid GPU programming model different, and programmers need tools to provide unified view of application's performance

# Performance Diagnosis of Hybrid Applications: Project Overview

- A benchmark suite for evaluating performance tools under development
- High level diagnostic metrics to convey most important performance trends (note: want them to work for CUDA and OpenCL)
- Visualizations to convey most important performance trends
- Funding for *Curriculum Innovation in Multicore Computing* → New PSU Course: General Purpose GPU Computing

# Graphic Processor Unit (GPU)



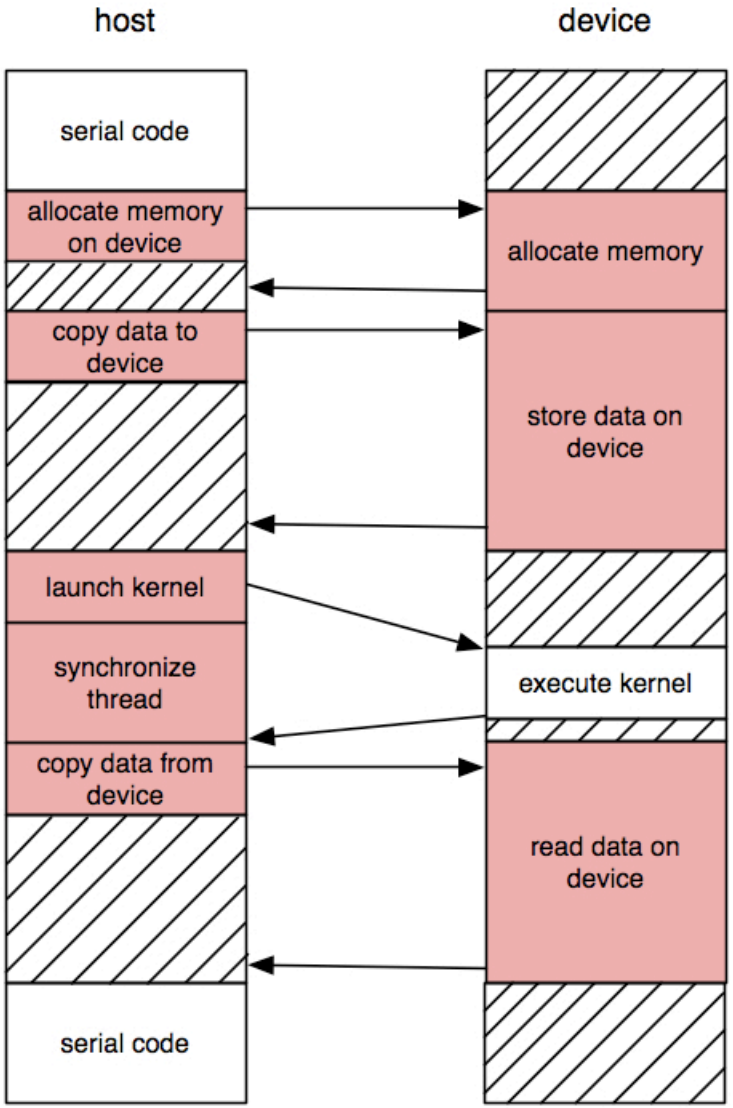

Scheduling unit: warp = 32 threads
Max per MP: 8 blocks, 32 warps

Nvidia Cuda Programming Guide, http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf

# GPU Programming


http://www.geek.com/wp-content/uploads/2009/03/xeon-processor.jpg

http://cc.doc.ic.ac.uk/projects/prj_axel/images/Tesla_c1060.png




Compute Cap.>= 2.0: streams

Compute Cap.>= 2.0: multiple kernels

# GPU Programming With CUDA

```
1 int main(int argc, char** argv) {
2     int a_host;
3     int b_host;
4     int answer;
5
6     a_host = thread_id;
7     b_host = 3;
8
9     answer = a_host + b_host;
10
11    return 0;
12  }
```

```
1 __global__ void add(int *a, int *b, int *answer_dev) {
2     *answer_dev = *a + *b;
3 }
4
5 int main(int argc, char** argv) {
6     int a_host;
7     int *a_dev;
8     int b_host;
9     int *b_dev;
10     int answer;
11     int *answer_dev;
12
13     a_host = thread_id;
14     b_host = 3;
15
16     cudaMalloc((void **) &answer_dev, sizeof(int));
17     cudaMalloc((void **) &a_dev, sizeof(int));
18     cudaMalloc((void **) &b_dev, sizeof(int));
19
20     cudaMemcpy(a_dev, &a_host, sizeof(int), cudaMemcpyHostToDevice);
21     cudaMemcpy(b_dev, &b_host, sizeof(int), cudaMemcpyHostToDevice);
22
23     dim3 dimGrid(1);
24     dim3 dimBlock(1,32);
25     add<<<dimGrid, dimBlock, 0>>>(a_dev, b_dev, answer_dev);
26
27     cudaThreadSynchronize();
28
29     cudaMemcpy(&answer, answer_dev, sizeof(int), cudaMemcpyDeviceToHost);
30
31     return 0;
32 }
```

define kernel

serial code

allocate memory

move data

launch kernel

move data

serial code

# A Simple Performance Tool Benchmark Suite

Inspired by APART Test Suite

Goals:

- Easily understandable behavior
- Used to check correctness of diagnosis

- Started with an existing single device implementation of matrix multiplication and extended it to support:
  - Multiple GPU devices
  - Overlap CPU and GPU computation
  - Use pinned memory
  - Use asynchronous memory transfers
- Based on *Programming Massively Parallel Processors: A Hands-on Approach* by Kirk and Hwu

# Naive Kernel

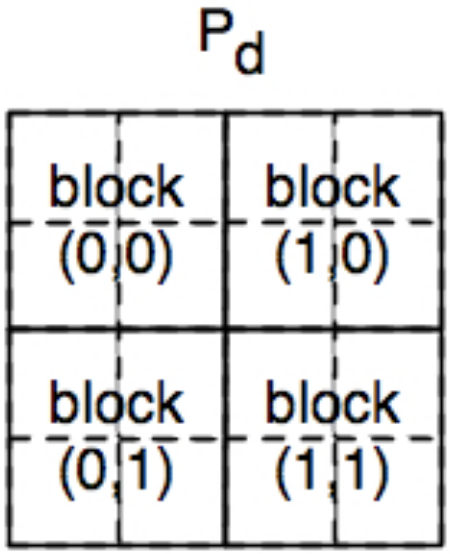
P = M x N [4x4]
d: data in global memory
ds: data in shared memory

- Matrix multiplication performed by calculating dot product of rows in M and columns in N
- Kernel configured to use one block of threads, so size of matrices limited by the maximum number of threads in a block
- In our case this was 16x16
  - Maximum number of threads in a block is 512
  - 16x16 is largest size that is a power of two

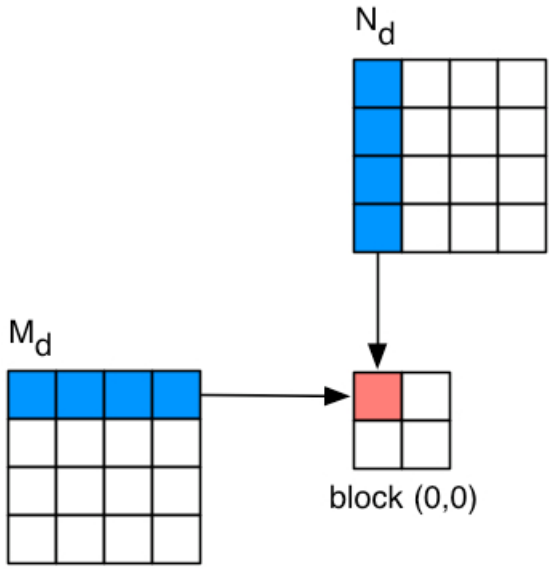
# Tiled Kernel

P = M x N [4x4]
d: data in global memory
ds: data in shared memory

- Breaks the $P_d$ matrix up into tiles
- Tiles the same size as a block



# Tiled Kernel



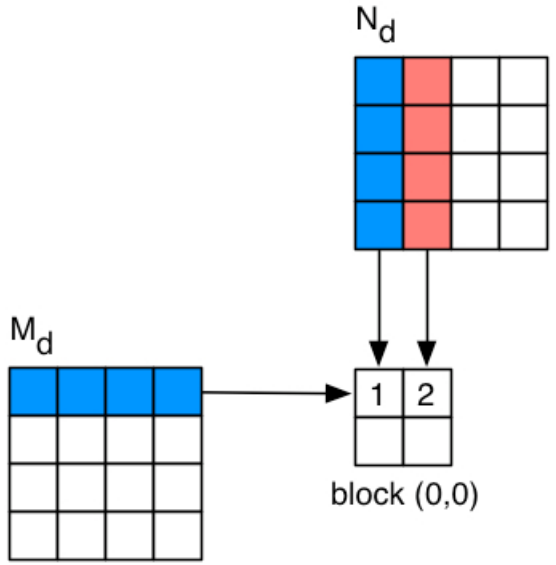

P = M x N [4x4]

d: data in global memory

ds: data in shared memory

# Tiled Plus Shared Memory Kernel

- Tiled kernel accesses data in global memory multiple times
- Tiled plus shared memory kernel uses shared memory to reduce global memory traffic
- Breaks computation up into phases
- Threads cooperatively load elements into shared memory




P = M x N [4x4]

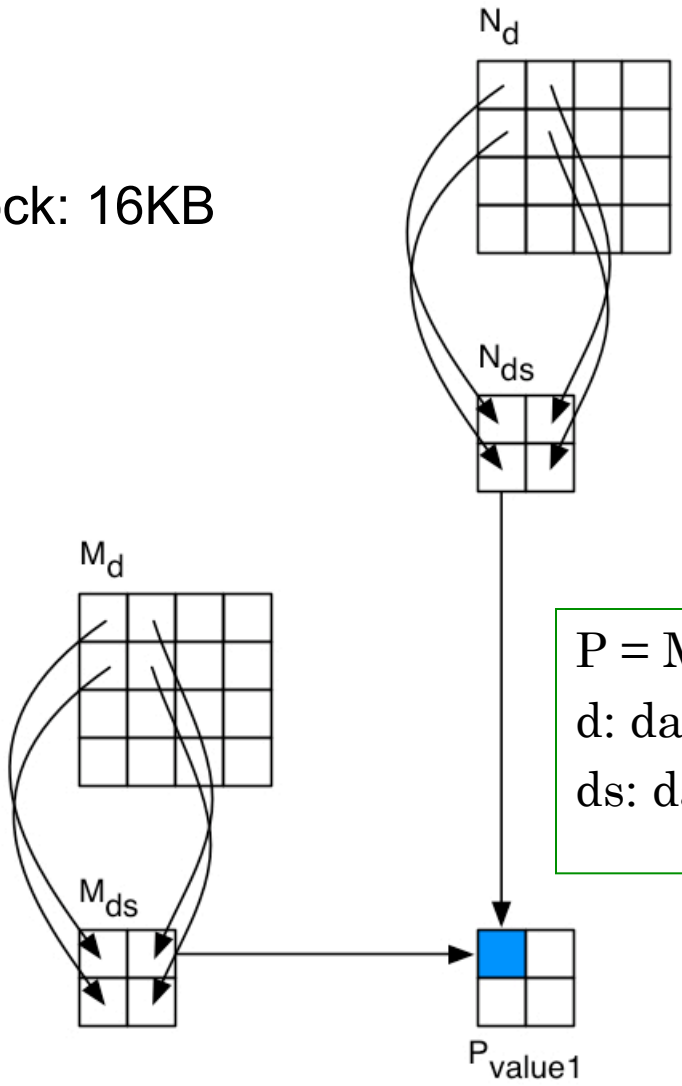
d: data in global memory

ds: data in shared memory

# Phase One

Global Memory: 4GB

Shared Memory per Block: 16KB




P = M x N [4x4]

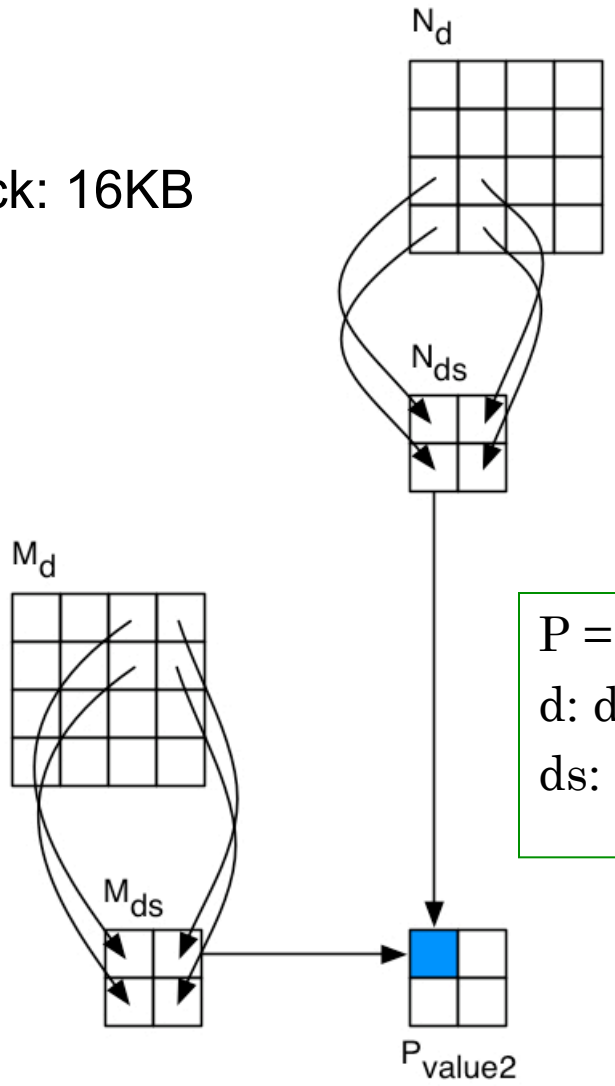
d: data in global memory

ds: data in shared memory

# Phase Two

Global Memory: 4GB

Shared Memory per Block: 16KB




P = M x N [4x4]

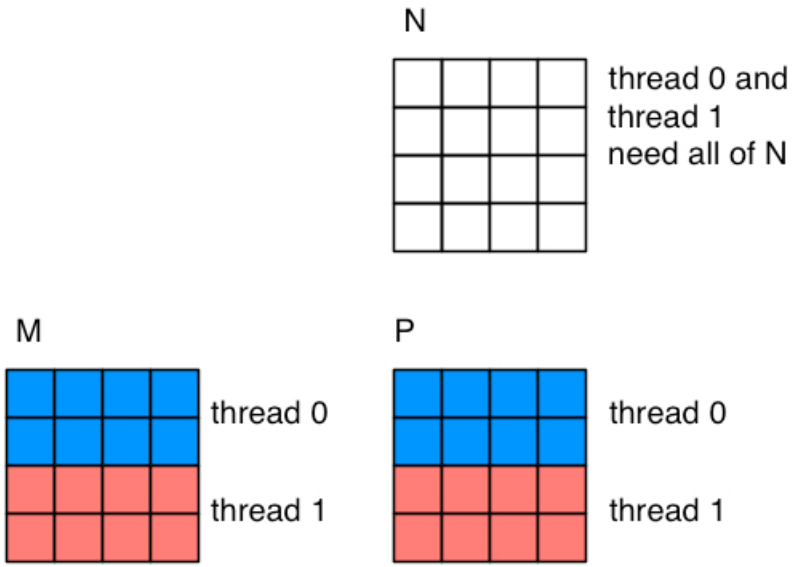
d: data in global memory

ds: data in shared memory

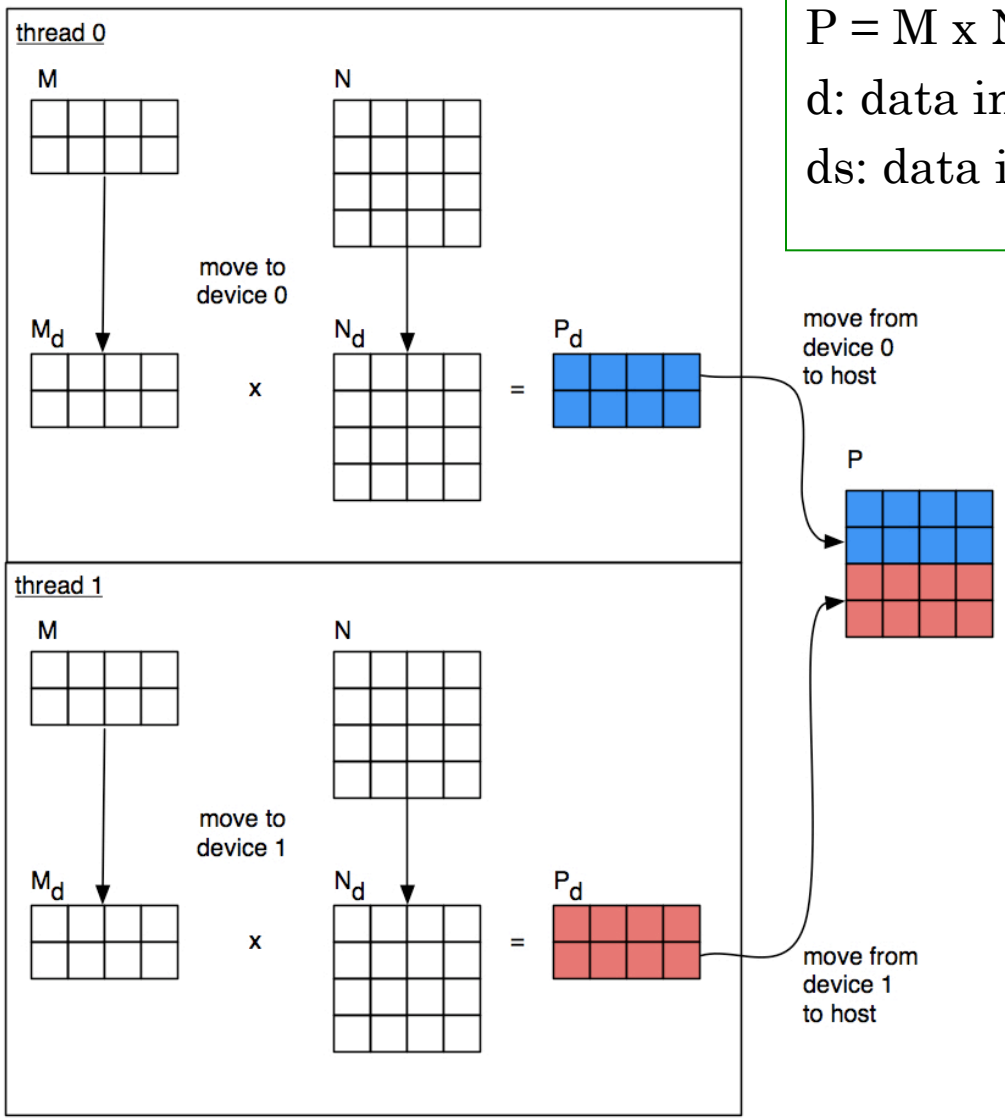
# Multiple Device Support

P = M x N [4x4]
d: data in global memory
ds: data in shared memory

- CUDA programming model requires at least one host thread per device
- We divide the work among multiple host threads.



# Multiple Device Support




P = M x N [4x4]

d: data in global memory

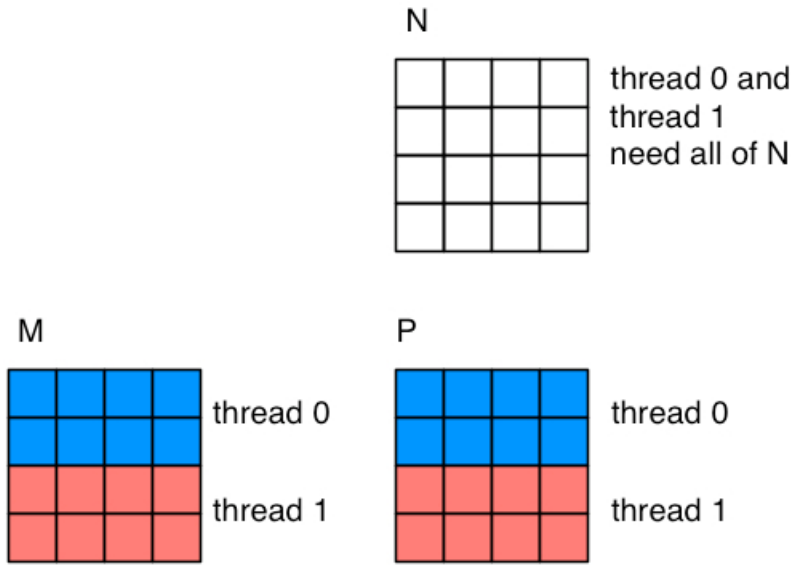
ds: data in shared memory

# Pinned Memory Optimizations

- Asynchronous memory transfers
  - Requires pinned memory
  - Non-blocking memory transfers for host

# Hybrid Matrix Multiplication

- Overlaps CPU and GPU computation
- Divides work same as multiple device version
  - One thread performs computation on device
  - Other thread performs computation on host



# Device Efficiency Metric

- Goal - indicate if the overhead of moving data to the device was justified
- Theoretical definition:  kernel compute time / kernel wall clock
- Kernel device time: amount of time kernel executes on device
- Device time: amount of time kernel and data movement functions execute on the device
- Minimum value: 0
- Maximum value: 1

$$DE = \frac{\text{kernel device time}}{\text{device time}}$$

# Device Utilization

- Goal - show how much of device's available computation is used
- Device time: amount of time kernel and data movement functions execute on the device
- Wall clock time: amount of time the application ran
- Minimum value: 0
- Maximum value: 1

$$DU = \frac{\text{device time}}{\text{wall clock time}}$$

# Experiments

Goals:

- Use matrix multiplication benchmark suite to study the behavior of the derived metrics
- What do the derived metrics tell us about a real scientific application ?

# Experimental Design - Instrumentation

CudaProf configured to gather:

- kernel device time: gputime for kernel functions
- device time: gputime for kernel and data movement functions

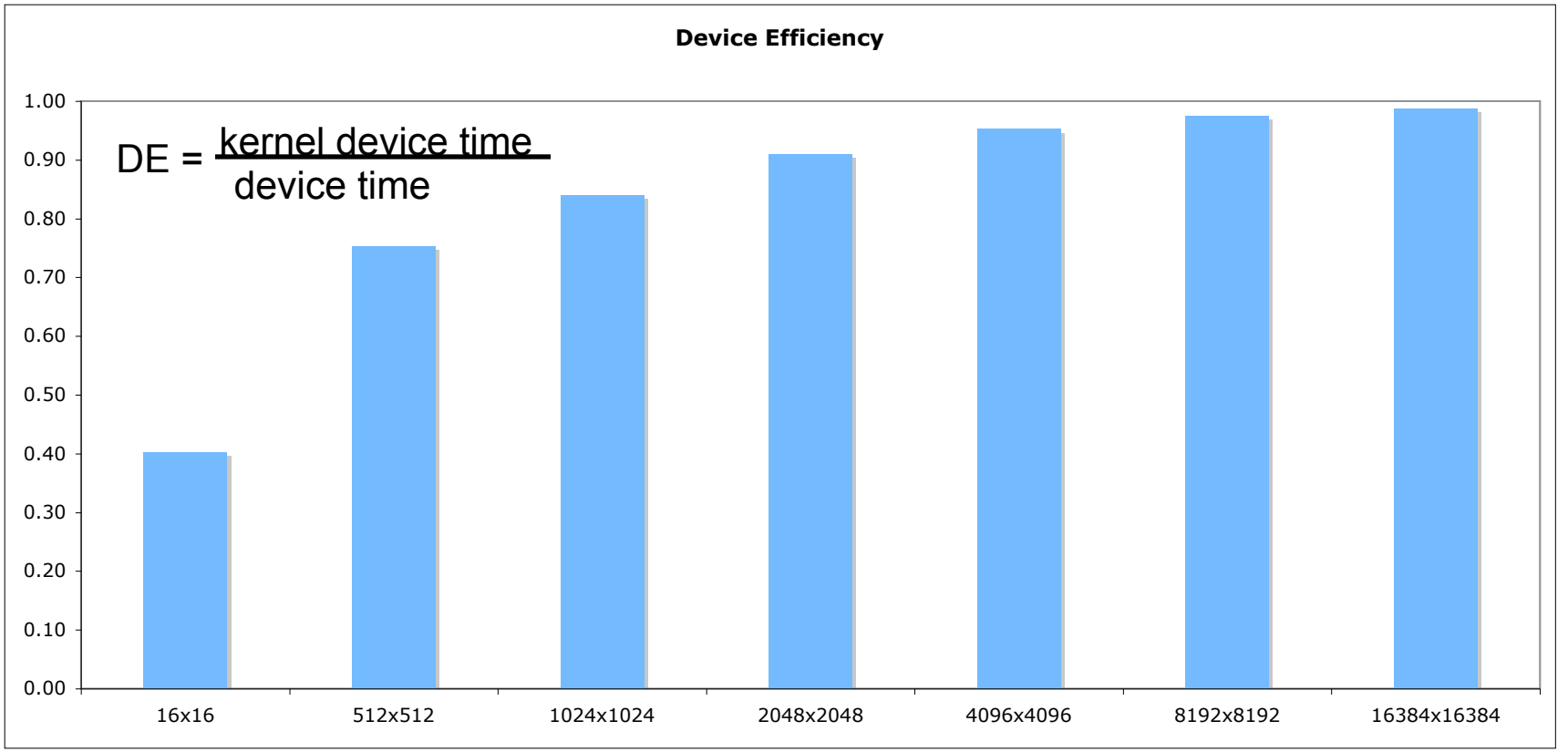
- wall clock time: external time utility
- Tesla C1060 (1 GPU), S1070 (4 GPUs): compute capability 1.3

$$DE = \frac{\text{kernel device time}}{\text{device time}} \qquad DU = \frac{\text{device time}}{\text{wall clock time}}$$

# Experimental Design - Matrix Multiplication

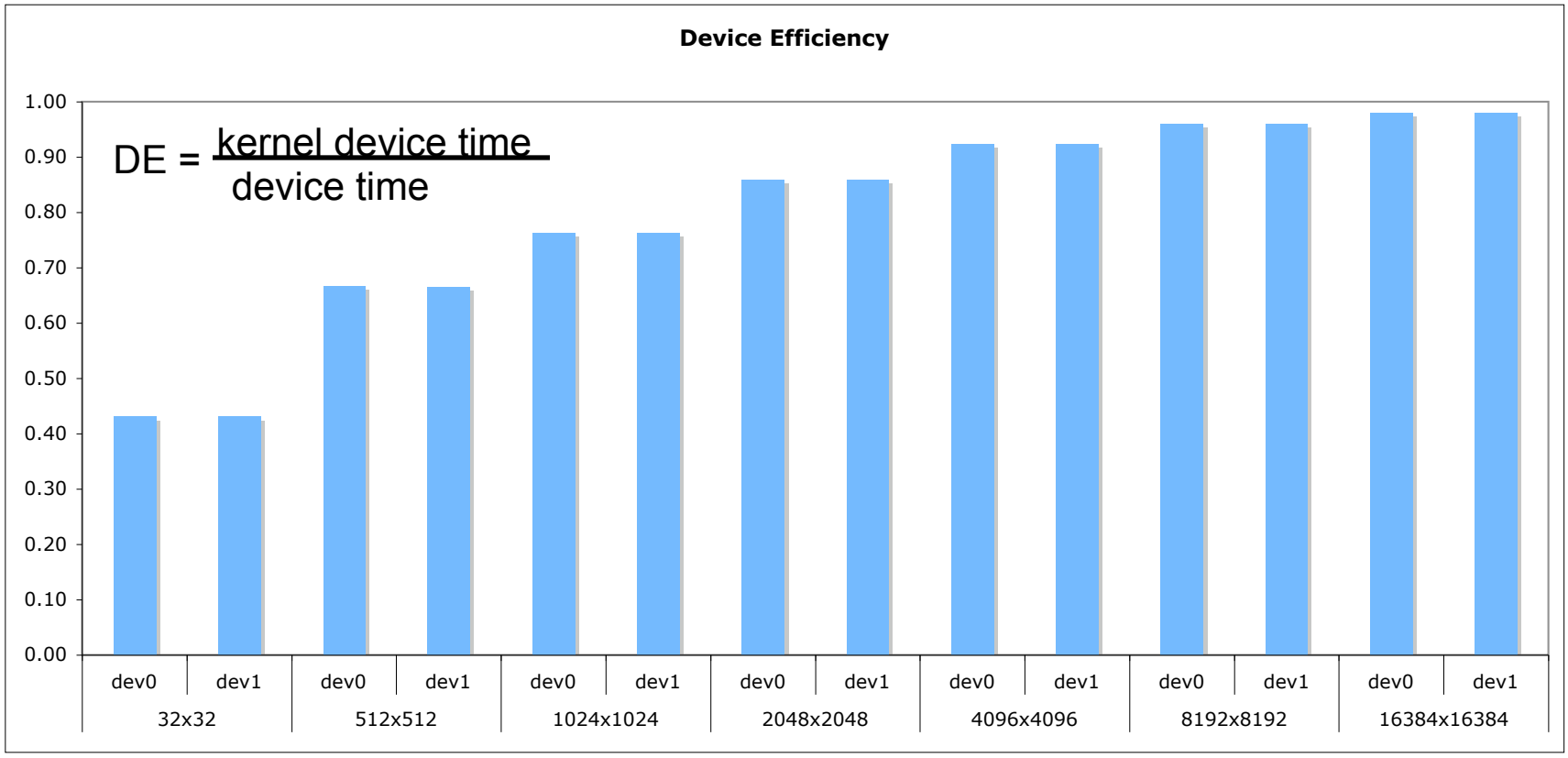
- Kernels:
  - Naive
  - Tiled
  - Tiled plus shared memory
- Memory optimizations:
  - Paged memory
  - Pinned memory
  - Pinned memory with asynchronous memory transfers
- Multiple devices
- Overlapped CPU and GPU computation
- Matrix sizes:
  - from 16x16 to 16,384x16,384

# Matrix Multiplication Case Study



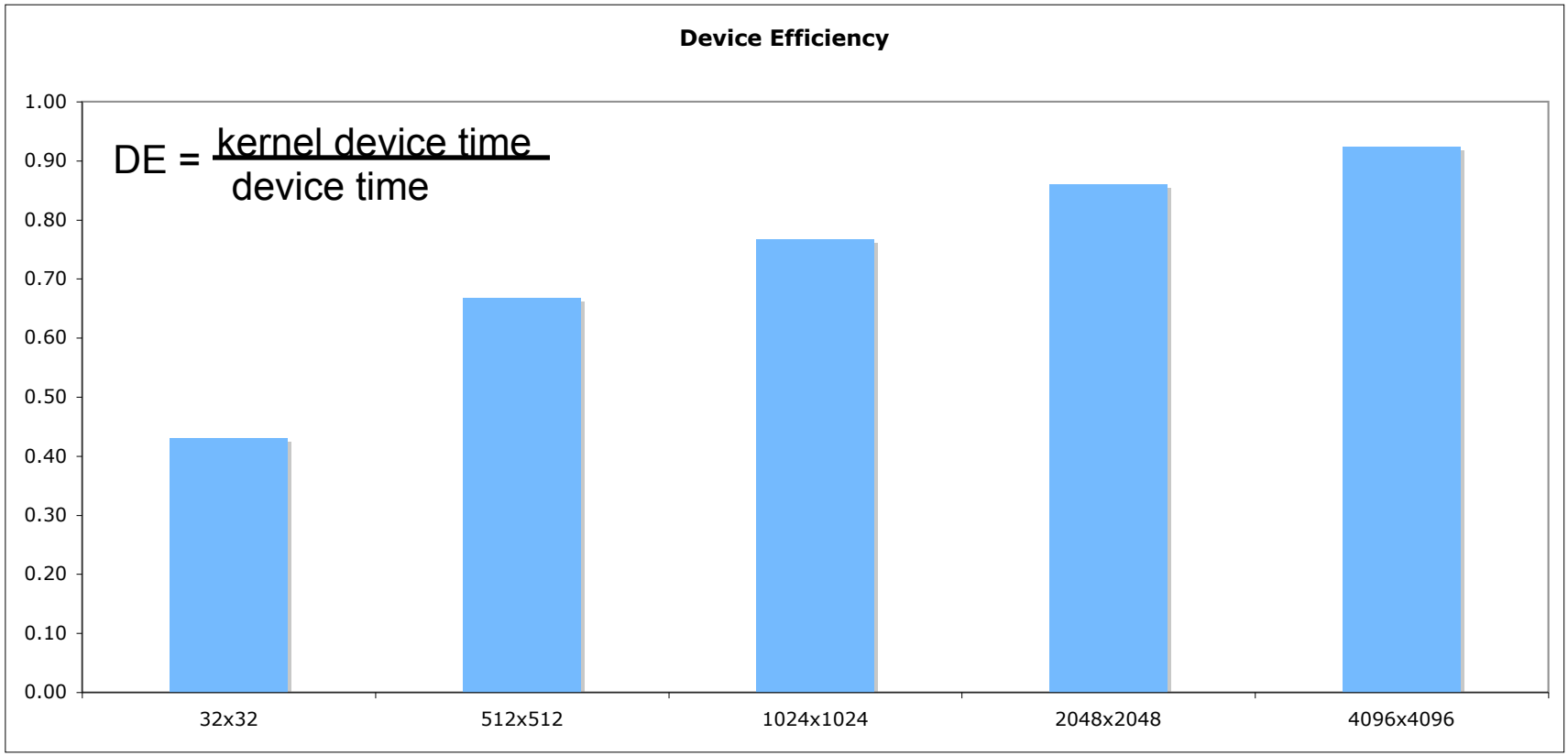

Matrix multiplication configured with: tiled+shared memory kernel, pinned memory, asynchronous memory transfers, single device

# Matrix Multiplication Case Study



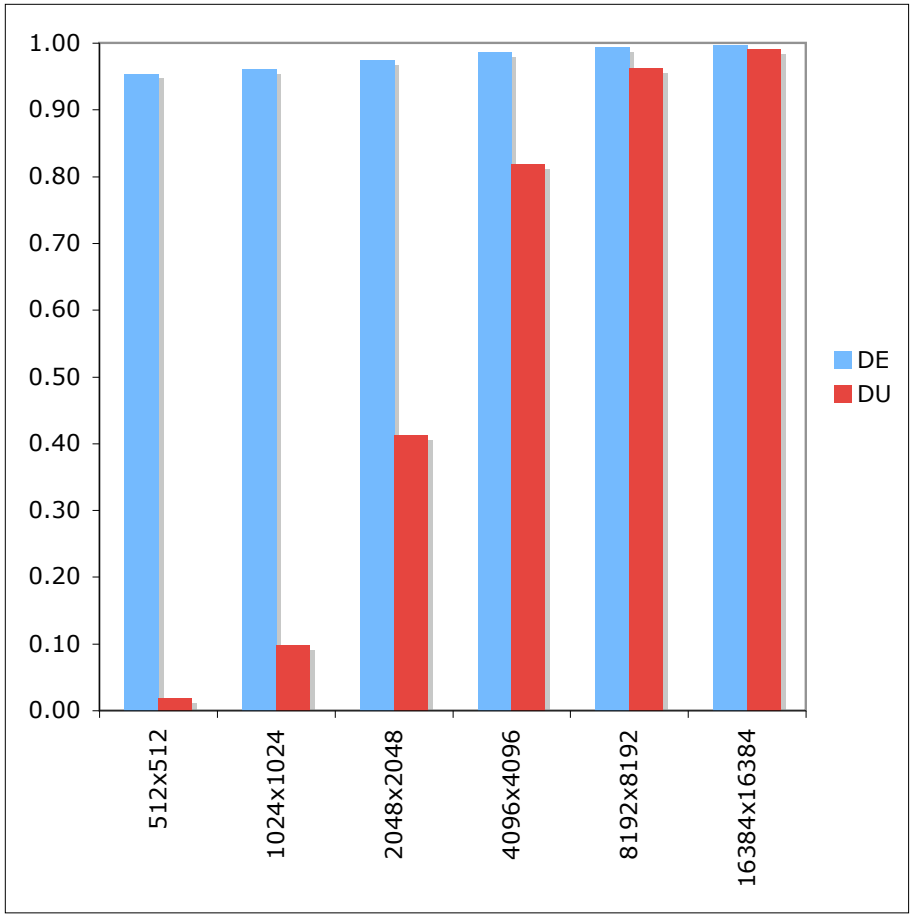

Matrix multiplication configured with: tiled+shared memory kernel, pinned memory, asynchronous memory transfers, two devices

# Matrix Multiplication Case Study

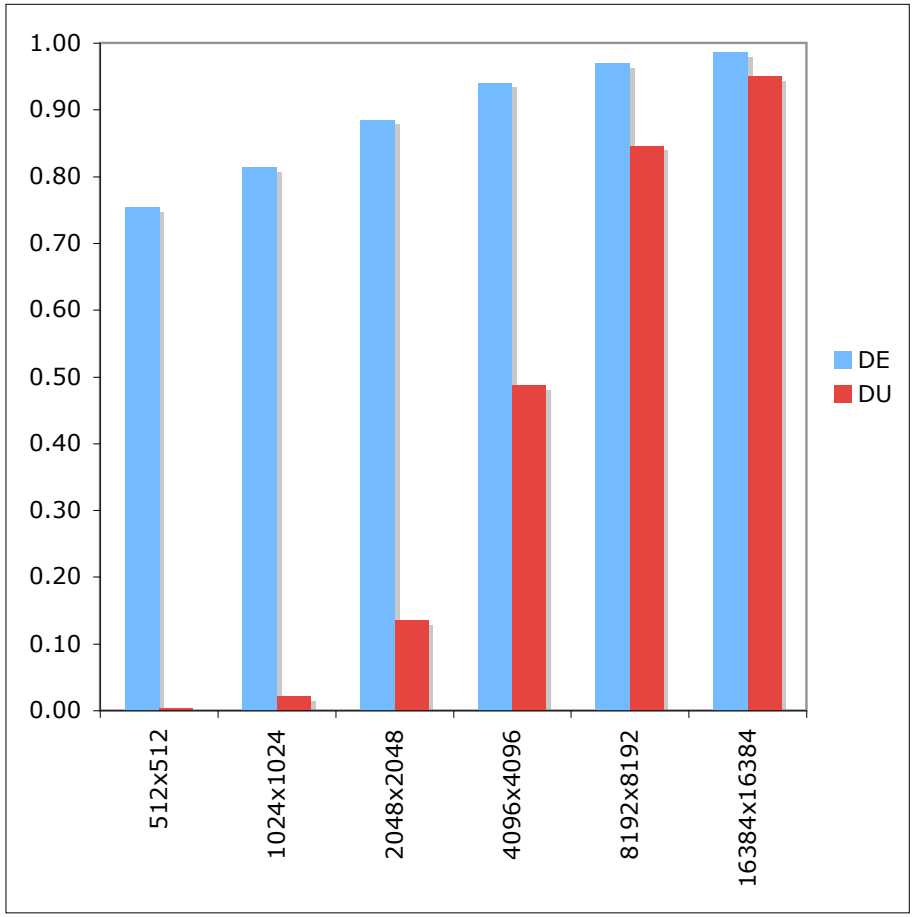



Matrix multiplication configured with: tiled+shared memory kernel, pinned memory, asynchronous memory transfers, overlapping CPU and GPU computation

# Matrix Multiplication Case Study




Matrix multiplication using tiled kernel, paged memory, single device.




Matrix multiplication using tiled plus shared memory kernel, paged memory, single device.

# Matrix Multiplication Case Study

| Matrix Size | Device Utilization | | |
|---|---|---|---|
| | 1 device | 2 devices | |
| | dev0 | dev0 | dev1 |
| 16x16 | 0.00 | 0.00 | 0.00 |
| 512x512 | 0.00 | 0.00 | 0.00 |
| 1024x1024 | 0.01 | 0.01 | 0.01 |
| 2048x2048 | 0.09 | 0.04 | 0.04 |
| 4096x4096 | 0.37 | 0.23 | 0.23 |
| 8192x8192 | 0.78 | 0.61 | 0.61 |
| 16384x16384 | 0.93 | 0.85 | 0.85 |

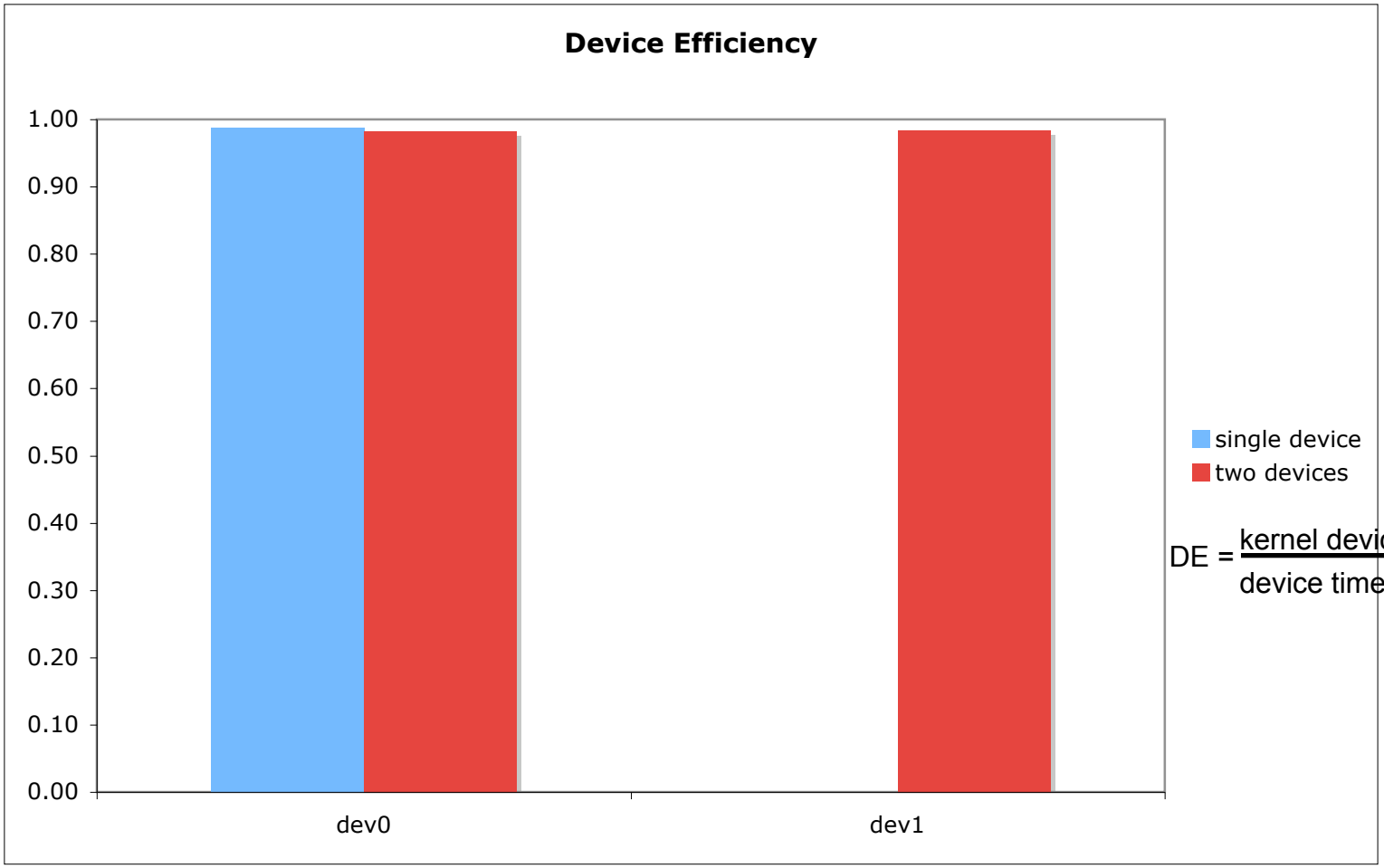
$$DU = \frac{\text{device time}}{\text{wall clock time}}$$

Matrix multiplication configured with: tiled+shared memory kernel, pinned memory, asynchronous memory transfers

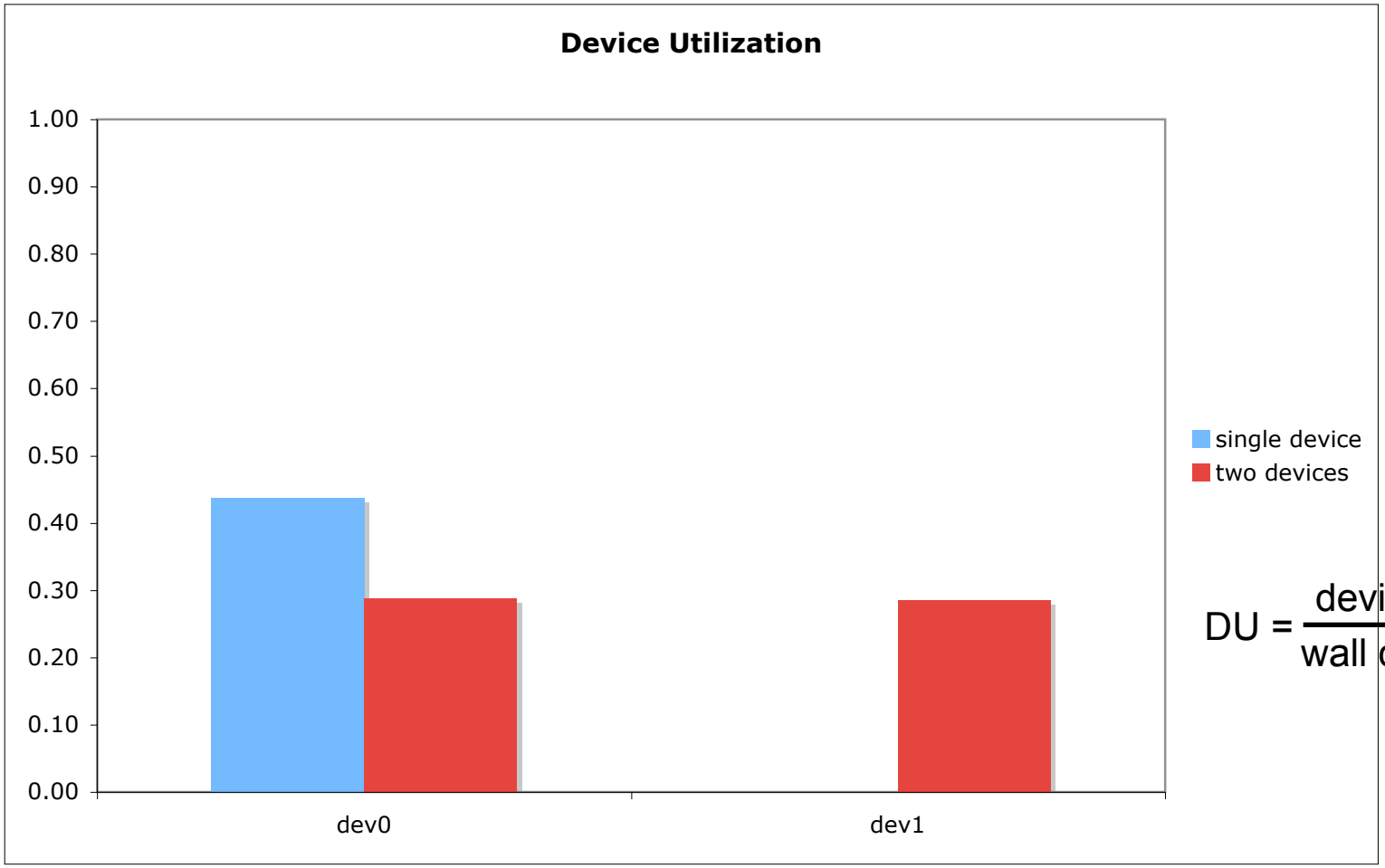
# Experimental Design - NAMD

- Are these metrics useful for full-scale application? Molecular dynamics simulator
- Extended by Phillips et al. to support GPUs
- Configured to simulate the Satellite Tobacco Mosaic Virus (STMV)
- Modified an existing simulation configuration

# NAMD Case Study



# NAMD Case Study




$$DU = \frac{\text{device time}}{\text{wall clock time}}$$

# Observations & Questions

- *Device efficiency* can be used to indicate performance in a hybrid environment, but doesn't reflect optimizations between kernels
- *Device utilization* can be used to indicate performance in a hybrid environment, but doesn't show how much an application *can* be accelerated
- Does device utilization matter?
- What about CPU utilization? Is CPU wait time "free"? FLOPs/watt?
- How can we visualize the asynchronous transfers and streams?
- Most common student questions:
  - How can I tell what the GPU is doing?? (perf.,scheduling)
  - How should I break down the problem? (blocks, streams)

# Acknowledgments and contact info


- *This work funded in part by National Science Foundation Award #1044973.*

- *Thanks to all students in "Performance Analysis of Heterogeneous Multicore Systems."*


- *Benchmark suite completed for CUDA and OpenCL.*


- *karavan@cs.pdx.edu*
- *www.cs.pdx.edu/~karavan*