RICE George R. Brown School of Engineering
Computer Science

Dynamic Task Parallelism with a GPU Work-Stealing Runtime System

Sanjay Chatterjee, Max Grossman,
Alina Sbîrlea, and Vivek Sarkar

Department of Computer Science
Rice University

Background

- As parallel programming enters the mainstream, the focus of programming models is primarily in multicore CPUs
- Many advantages to using heterogeneous hardware
  - Performance
  - Power consumption per FLOP
  - Price per FLOP
- Tradeoff: application performance vs. learning curve

Background

- GPUs are a promising example of heterogeneous hardware
  - Hundreds of cores, split among stream multiprocessors (SMs)
  - High memory bandwidth to global memory
  - Low power consumption

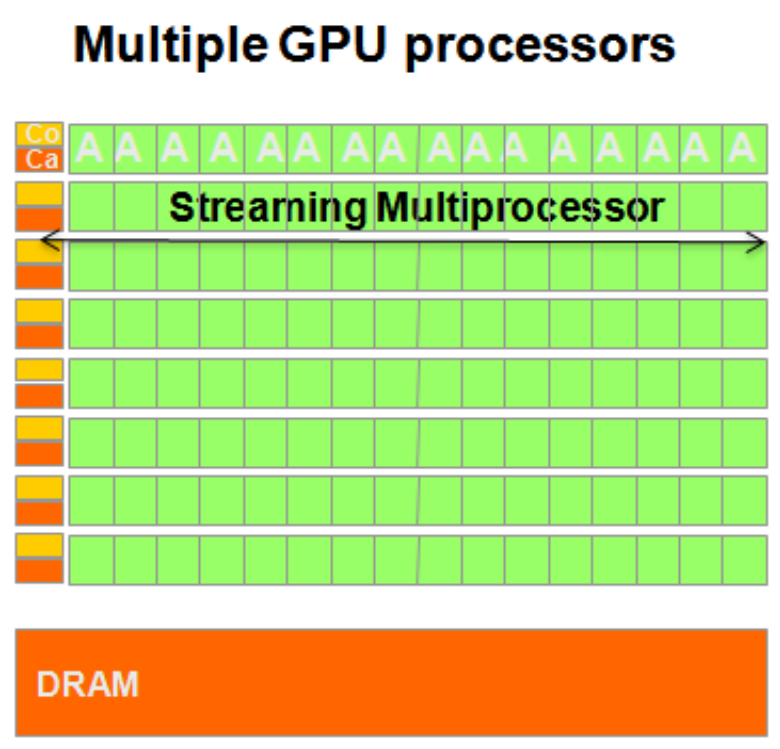
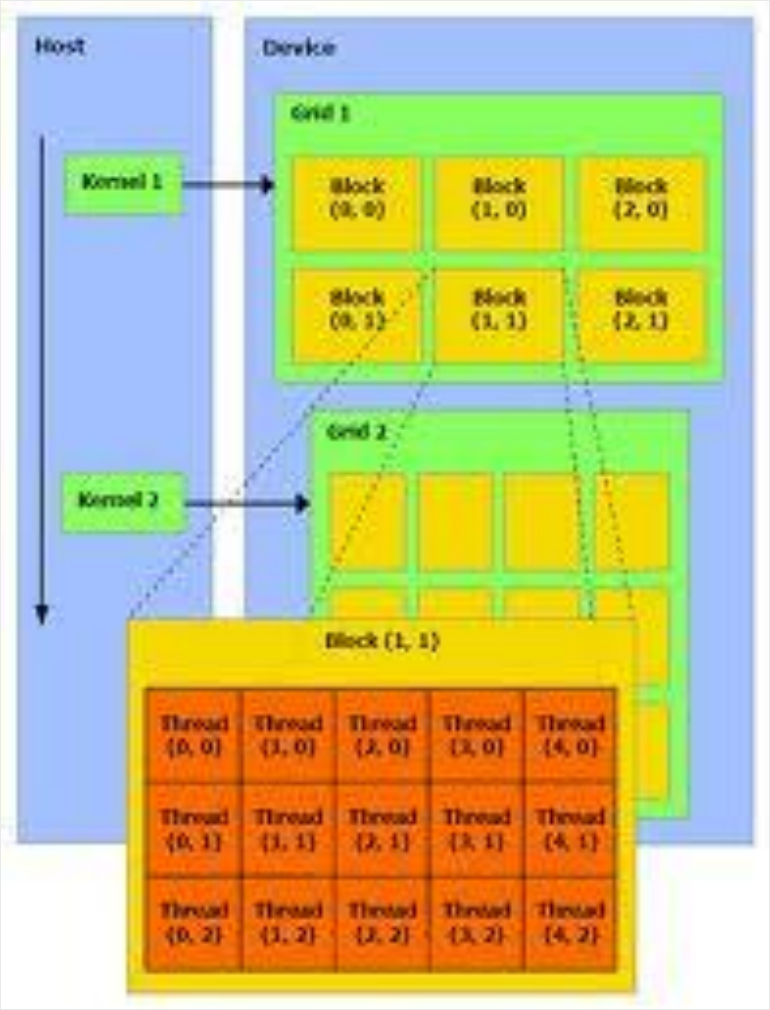


Figure source: David B. Kirk and Wen-mei W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

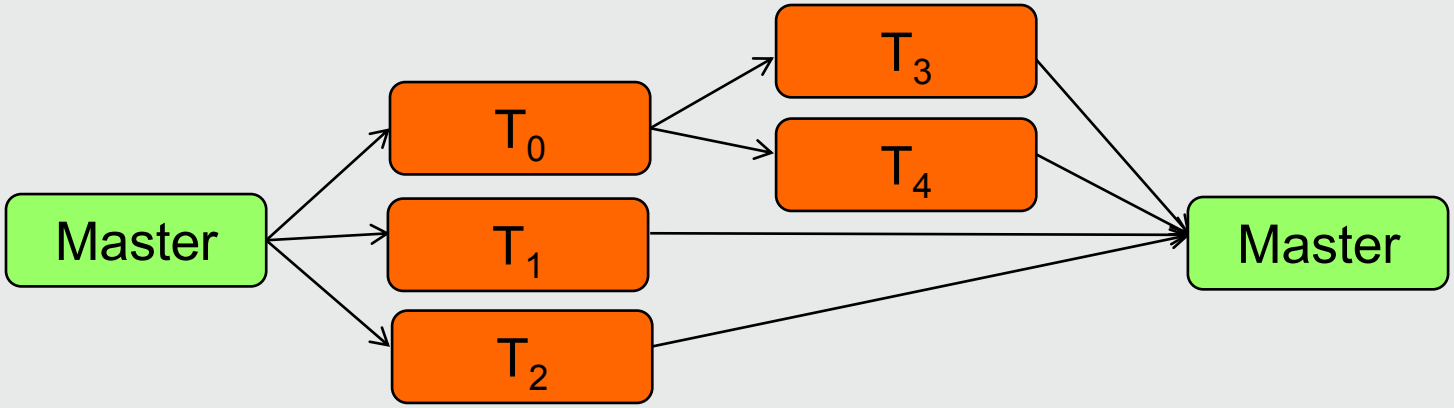
Background

- NVIDIA's CUDA makes general purpose programming on GPUs possible, but not simple
  - Explicit/static device memory management
  - Understanding of the underlying hardware necessary for evaluating performance tradeoffs
  - Static thread pool within a device kernel



Problem

- Parallel programmers are more familiar with dynamic task parallelism on multicore CPUs than static data parallelism of GPUs



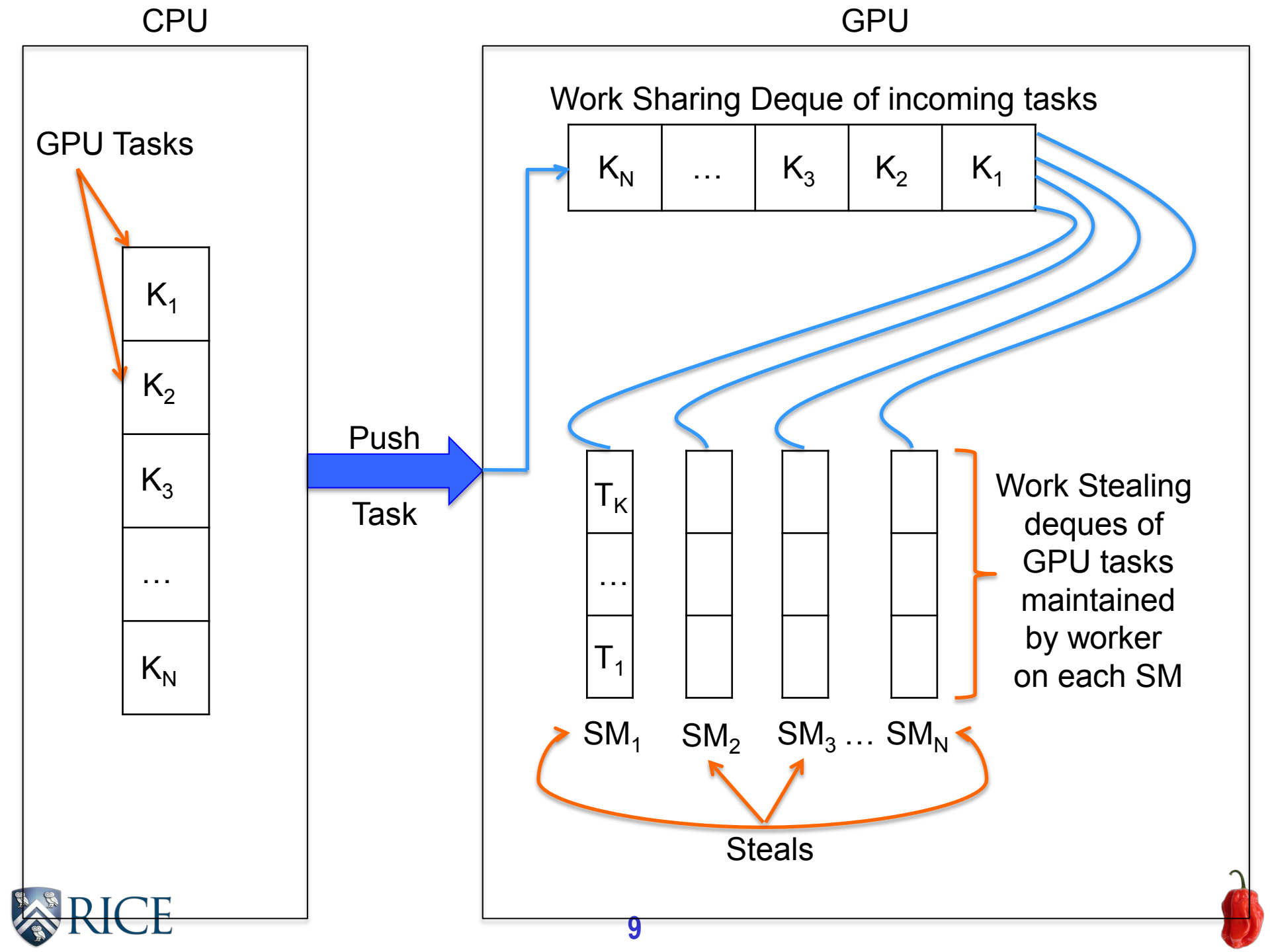
- But many applications could benefit from the use of GPUs

Solution

- Build a hybrid work sharing/stealing runtime on a single device
- Use this runtime to facilitate:
  - Dynamic task parallelism
  - Automatic device memory management
  - Increased programmability
  - Transparent multi-GPU execution

Design

- Persistent runtime kernel on the device
- Host pushes new tasks to the device through a work sharing deque shared among SMs on the device
- SMs on the same device share tasks through a work stealing deque assigned to each SM
  - Batched steals
- Load balancing is done at the thread block granularity, so tasks execute on worker blocks rather than worker threads


CPU
GPU
GPU Tasks
K1
K2
K3
…
KN
Push
Task
Work Sharing Deque of incoming tasks
KN
…
K3
K2
K1
TK
…
T1
Work Stealing deques of GPU tasks maintained by worker on each SM
SM1
SM2
SM3
…
SMN
Steals

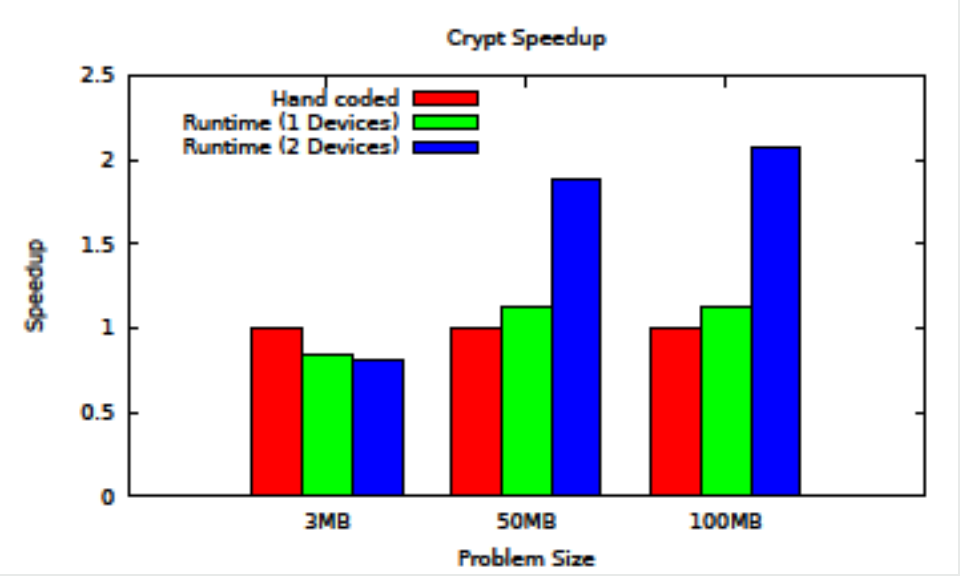
Design

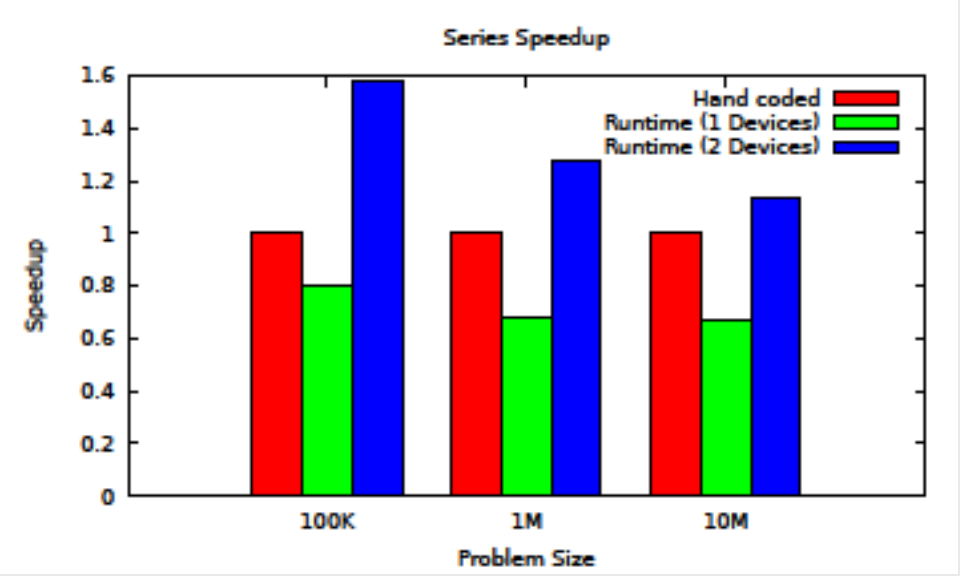
- Runtime API provides simpler access to device
  - `insert_task(...)`
  - `get_data(...)`
  - `init_runtime(...)`
  - `finish_device()`

Evaluation

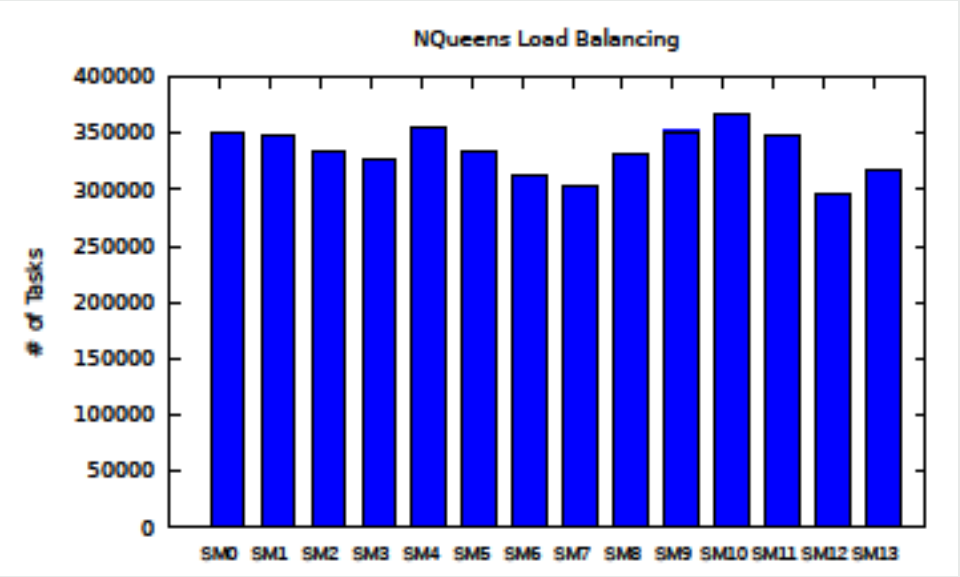
- Evaluation was performed using a variety of benchmarks to study different properties
  - Crypt (Java Grande Forum Benchmark Suite)
  - Series (Java Grande Forum Benchmark Suite)
  - Dijkstra (Lastras-Montano et. al., “Dynamic Work Scheduling for GPU Systems”)
  - Nqueens (BOTS)
  - Unbalanced Tree Search (OSU)
- Code comparison

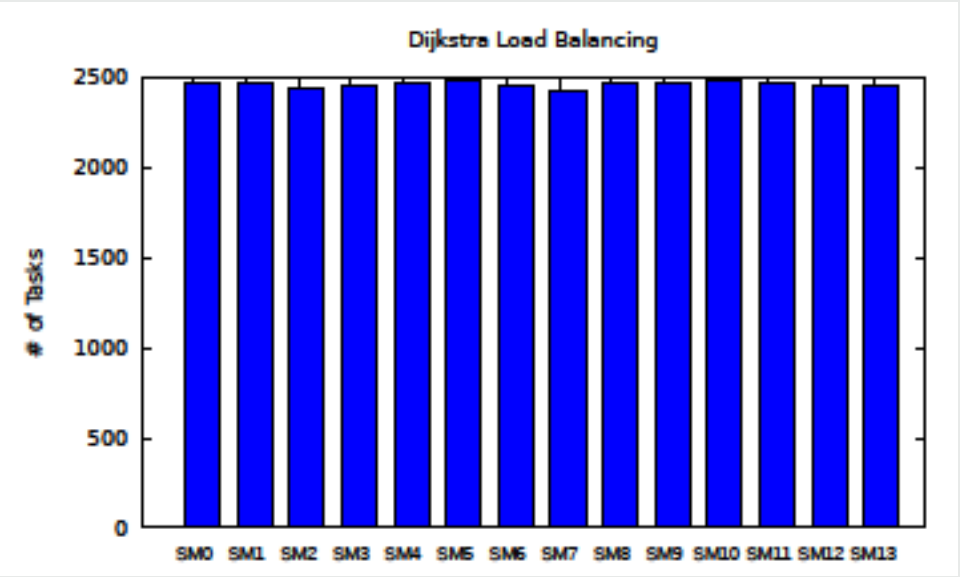
Data Parallel Performance Evaluation



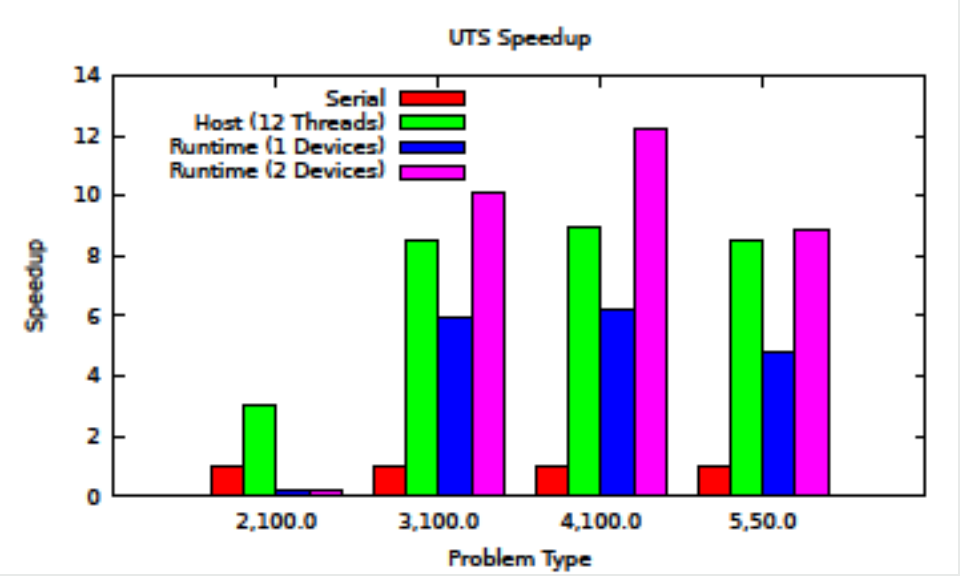


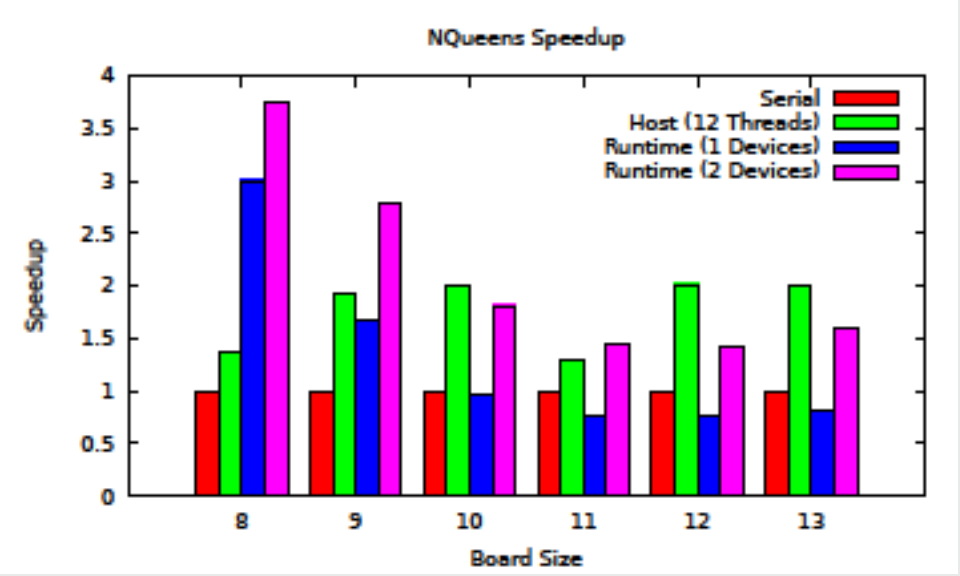
Load Balance Evaluation





Task Parallel Performance Evaluation





Code Comparison

CUDA

```
cudaMalloc((void **)&d_Z, sizeof(int) * 52);
cudaMalloc((void **)&d_DK, sizeof(int) * 52);
cudaMalloc((void **)&d_plain, mem_size);
cudaMalloc((void **)&d_crypt, mem_size);
cudaMemcpy(d_crypt, crypt, mem_size,
    cudaMemcpyHostToDevice);
cudaMemcpy(d_plain, plain1, mem_size,
    cudaMemcpyHostToDevice);
cudaMemcpy(d_Z, Z, sizeof(int) * 52,
    cudaMemcpyHostToDevice);
cudaMemcpy(d_DK, DK, sizeof(int) * 52,
    cudaMemcpyHostToDevice);

cryptCUDAKernel<<<threads_per_block,
    blocks_per_grid>>>(d_plain, d_Z, d_crypt,
    mem_size);
cryptCUDAKernel<<<threads_per_block,
    blocks_per_grid>>>(d_crypt, d_DK,
    d_plain,  mem_size);
cudaMemcpy(crypt, d_crypt, mem_size,
    cudaMemcpyDeviceToHost);
cudaMemcpy(plain2, d_plain, mem_size,
    cudaMemcpyDeviceToHost);
```

Runtime

```
InitRuntime(...)
... Task construction code ...
for(i = 0; i < n_tasks; i++) {
    insert_task(tasks+i);
}
for(i = 0; i < n_tasks; i++) {
    get_data(plain2+(i * mem_per_task));
}
finish_device_tasks();
```

- Simpler code to autogenerate when paired with a parallel compiler

Conclusions

- GPU work stealing runtime which supports dynamic task parallelism, on hardware intended for data parallelism
- Demonstrated effectiveness of work stealing deques in dynamically distributing work between SMs
- Enabled task parallel applications on GPUs
- Demonstrated reasonable (though not stellar) performance for both data parallel and task parallel applications

Future Work

- Investigate bottlenecks of the runtime or of task parallel applications using the runtime (i.e. UTS, NQueens)
- Integrate this runtime with the Habanero C and Concurrent Collections (CnC) programming systems under development at Rice University
  - The GPU work-stealing runtime is a standalone tool which can be integrated with a programming model
  - Facilitate programmer access to heterogeneous hardware
  - Hand coded integration has already been demonstrated

Future Work

- The GPU work-stealing runtime is a standalone tool which can be integrated with a programming model in order to provide a user friendly interface
- Tag puts analogous to `insert_task` on the device
- Facilitate programmer access to heterogeneous hardware
- Hand coded integration has already been demonstrated, with the next step being auto generation

Questions?