Tuned and Wildly Asynchronous Stencil Kernels for Hybrid CPU/GPU Systems

Mickey Ju

May 12, 2011
Jacobi’s method for the 2D Poisson Equation

Discrete Approximation

\[
\text{for } t \leftarrow 1 \text{ to } T \text{ do} \\
\quad \text{for } i \leftarrow 1 \text{ to } n \text{ do} \\
\quad \quad \text{for } j \leftarrow 1 \text{ to } n \text{ do} \\
\quad \quad \quad U_{i,j}^{t+1} \leftarrow \frac{1}{4} \cdot \left( U_{i+1,j}^t + U_{i-1,j}^t + U_{i,j+1}^t + U_{k,j-1}^t + h^2 \cdot F_{i,j} \right) \\
\quad \quad \text{end for} \\
\quad \text{end for} \\
\text{end for}
\]
CPU Baselines

- Vectorization, verified by inspecting the assembly code
- Bind threads to cores using the Linux affinity scheduling routines
- Allocate the per-thread data blocks in the memory of the closest processor socket
- How to get the ghost-cells from other processors?
Naive GPU Implementation

- Used only global device memory
- Suffers from non-coalesced memory accesses? (Padding to solve it?)

Copy the grid from host to the device
for $t \leftarrow 1$ to $T$ do
  Invoke GPU kernel
  (Implicit) Synchronize host and device
  Logically swap active grid
end for
Copy results from device to host
GPU Implementation Using Shared Memory

- Reduces the accesses to global memory
- Stores only one grid in global memory
- Applies loop-unrolling
- Questions
Hybrid Implementation

- Expects the overlapping to help
- Needs data-transfer overhead to be low
- Same idea can be applied to multi-GPU implementation

assign rows 1 to $s$ to CPUs
assign rows $s + 1$ to $n$ to the GPUs
for $t ← 1$ to $T$ do
  GPU part and CPU part
  Exchange data
end for
Async0

- Transfers the dependent elements back to global memory
- Loads the ghost cells from global memory
- Can work totally in shared memory
- Drops the implicit synchronizations between kernel calls
- Introduces more synchronizations between threads

\[
\text{for } v \leftarrow 1 \text{ to } \alpha/2 \text{ do}
\]

- Compute 1 iteration in $B_1$, writing to $B_2$
  - \text{syncthreads()}
- Write penultimate fringe from $B_2$ to global memory
  - \text{syncthreads()}
- Fetch fringe elements from device memory to $B_2$
  - \text{syncthreads()}
- Compute 1 iteration in $B_2$, writing to $B_1$
  - \text{syncthreads()}
- Write penultimate fringe from $B_1$ to global memory
  - \text{syncthreads()}
- Fetch fringe elements from device memory to $B_1$
  - \text{syncthreads()}

\text{end for}

Compute 1 iteration in $B_1$
Async1

- Removes some synchronizations between threads

\[
\text{for } v \leftarrow 1 \text{ to } \alpha/2 \text{ do}
\]
\[
\text{Compute 1 iteration in } B_1, \text{ writing to } B_2
\]
\[
\text{Write penultimate fringe from } B_2 \text{ to global memory}
\]
\[
\text{Fetch fringe elements from device memory to } B_2
\]
\[
\_\_\_\_\text{syncthreads()}
\]
\[
\text{Compute 1 iteration in } B_2, \text{ writing to } B_1
\]
\[
\text{Write penultimate fringe from } B_1 \text{ to global memory}
\]
\[
\text{Fetch fringe elements from device memory to } B_1
\]
\[
\_\_\_\_\text{syncthreads()}
\]
\[
\text{end for}
\]
\[
\text{Compute 1 iteration in } B_1
\]
Async2

- Does not transfer elements on border back and forth

\[
\begin{align*}
\text{for } v & \leftarrow 1 \text{ to } \alpha/2 \text{ do} \\
& \quad \text{Compute 1 iteration in } B_1, \text{ writing to } B_2 \\
& \quad \quad \_\_\text{syncthreads()} \\
& \quad \text{Compute 1 iteration in } B_2, \text{ writing to } B_1 \\
& \quad \quad \_\_\text{syncthreads()} \\
\text{end for} \\
& \text{Compute 1 iteration in } B_1
\end{align*}
\]
Async3

- Eliminates all synchronizations between threads
- Uses only one array in kernel

\[
\text{for } \nu \leftarrow 1 \text{ to } \alpha \text{ do} \\
\text{Compute 1 iteration in } B, \text{ writing to } B \\
\text{Write penultimate fringe from } B \text{ to global memory} \\
\text{Fetch fringe elements from device memory to } B \\
\text{end for} \\
\text{Compute 1 iteration in } B
\]