Lecture 12
CSE 260 – Parallel Computation
(Fall 2015)
Scott B. Baden

Stencil methods
Announcements

- Weds office hours (11/4): 3:30 to 4:30
- For remainder of quarter: 3:30 to 5:30
Today’s lecture

• Stencil methods on the GPU
• Aliev Panfilov Method (A3)
• CUDA Implementation
Recapping from last time

- The warp scheduler dynamically reorders instructions to avoid pipeline stalls and other hazards
- We use scoreboard to keep track of instruction progress and available resources
- Branches serialize execution within a warp
- Warp aware coding can avoid divergence
Warp aware summation

For next time: complete the code so it handles global data with arbitrary N

reduceSum <<<N/512,512>>> (x,N)
Stencil methods

- Many physical problems are simulated on a uniform mesh in 1, 2 or 3 dimensions
- Field variables defined on a discrete set of points
- A mapping from ordered pairs to physical observables like temperature and pressure
- Important applications
  - Differential equations
  - Image processing
Motivating App: Digital Image Processing

- We represent the image in terms of *pixels*
- In color image, each pixel can contain 3 colors: RGB

\[
\begin{array}{cccccccccc}
\text{Red} & & & & & & & & & \\
\text{Green} & & & & & & & & & \\
\text{Blue} & & & & & & & & & \\
\end{array}
\]
Image smoothing algorithm

- An important operation is called image smoothing
- Replace each pixel by the average of its neighbors
- Repeat as many times as necessary

\[
\text{while not smooth enough do:} \\
\quad \text{for } (i,j) \text{ in } 0:\text{N}-1 \times 0:\text{N}-1 \\
\quad \quad I_{\text{new}}[i,j] = \left( I[i-1,j] + I[i+1,j] + I[i,j-1] + I[i, j+1] \right) / 4 \\
\quad I = I_{\text{new}}
\]

end while
A motivating app from biomedical computing

• Model signal propagation in cardiac tissue using the *Aliev-Panfilov method*
  - Demonstrates complex behavior of spiral waves that are known to cause life-threatening situations

• Reaction-diffusion system of equations
  - Reactions are the cellular exchanges of certain ions across the cell membrane during the cellular electrical impulse

• Our simulation has two state variables
  - *Transmembrane potential*: \( e \)
  - *Recovery of the tissue*: \( r \)
The Aliev-Panfilov Model

• Two parts
  ▶ 2 Ordinary Differential Equations
    • Kinetics of reactions occurring at every point in space
  ▶ Partial Differential Equation
    • Spatial diffusion of reactants

• A differential equation is a set of equations involving derivatives of a function (or functions), and specifies a solution to be determined under certain constraints

• Constraints often specify boundary conditions or initial values that the solution must satisfy

\[
\frac{\partial e}{\partial t} = \delta \nabla^2 e - ke(e-a)(e-1) - cr, \quad \text{on } \Omega_I,
\]

\[
\frac{\partial r}{\partial t} = - \left[ \varepsilon + \frac{\mu_1 r}{\mu_2 + e} \right] [r + ke(e-b-1)], \quad \text{on } \Omega_I,
\]

\[
\bar{n} \cdot \delta \nabla e = 0 \text{ on } \partial \Omega, \quad \text{and} \quad (e, r)_{t=0} = (e(\cdot, 0), r(\cdot, 0)),
\]
Discrete approximations

- The functions appearing in the differential equations are continuous
- On the computer, we can represent discrete values only
- We will approximate the solution on the mesh using the method of finite differences
- We use first-order explicit numerical scheme
- For example, we represent a 2nd derivative as

\[ u'' \approx \frac{(u(x+h) - 2u(x) + u(x-h))}{h^2} \]

\[ = \frac{(u[i+1] - 2u[i] + u[i+1])}{h^2} \]
Computational loop of the cardiac simulator

- **ODE solver:**
  - No data dependency, trivially parallelizable
  - Requires a lot of registers to hold temporary variables

- **PDE solver:**
  - Jacobi update for the 5-point Laplacian operator.
  - Sweeps over a uniformly spaced mesh
  - Updates voltage to weighted contributions from the 4 nearest neighbors updating the solution as a function of the values in the previous time step

For a specified number of iterations, using supplied initial conditions repeat

\[
\text{for (j=1; j < m+1; j++)} \quad \text{// PDE SOLVER}
\]
\[
\text{for (i=1; i < n+1; i++)}
\]
\[
E[j,i] = E_p[j,i] + \alpha (E_p[j,i+1]+E_p[j,i-1]-4*E_p[j,i]+E_p[j+1,i]+E_p[j-1,i]);
\]
\[
\text{for (j=1; j < m+1; j++)} \quad \text{// ODE SOLVER}
\]
\[
\text{for (i=1; i < n+1; i++)}
\]
\[
E[j,i] += -dt*(kk*E[j,i]*(E[j,i]-a)*(E[j,i]-1)+E[j,i]*R[j,i]);
\]
\[
R[j,i] += dt*(\varepsilon+M1* R[j,i]/(E[j,i]+M2))*(-R[j,i]-kk*E[j,i]*(E[j,i]-b-1));
\]

\text{swap } E_p \text{ and } E

\text{End repeat}
Performance of the simulator on Sorken

- Use valgrind to obtain cache performance, down to the source line (see Getting Started with Bang for instructions, same as for Sorken)

make –j N runs parallelizes the make on N threads

- Program runs very slowly so be sure to use a qlogin node

valgrind --tool=cachegrind ./apf -n 256 -i 2000
cg_annotate --auto=yes cachegrind.out.7090 > Report.txt
Where is the time spent? [Provided Code]

- Loops are unfused

```
I1 cache: 32768 B, 64 B, 8-way associative
D1 cache: 32768 B, 64 B, 8-way associative
LL cache: 20971520 B, 64 B, 20-way associative
Command: ./apf -n 256 -i 2000
```

```
<table>
<thead>
<tr>
<th></th>
<th>Ir</th>
<th>Ilmr</th>
<th>ILmr</th>
<th>Dr</th>
<th>Dlmr</th>
<th>DLmr</th>
<th>Dw</th>
<th>Dlmw</th>
<th>DLmw</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4.451B</td>
<td>2,639</td>
<td>2,043</td>
<td>1,381,173,237</td>
<td>50,592,465</td>
<td>7,051</td>
<td>3957M</td>
<td>16,794,937</td>
<td>26,115</td>
</tr>
<tr>
<td>PROGRAM TOTALS</td>
<td>Dr</td>
<td>Dlmr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>------</td>
<td>------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,380,464,019</td>
<td>50,566,007</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
solve.cpp:solve( ...)
.
.
.
// Fills in the TOP Ghost Cells
10,000 1,999 for (i = 0; i < n+2; i++)
516,000 66,000 E_prev[i] = E_prev[i + (n+2)*2];
// Fills in the RIGHT Ghost Cells
10,000 0 for i = (n+1); i < (m+2)*(n+2); i+=(m+2)
516,000 504,003 E_prev[i] = E_prev[i-2];
// Solve for the excitation, a PDE
1,064,000 8,000 for(j =innerBlkRowStartIndx;j<=innerBlkRowEndIndx; j+=(m+)){
0 0 E_prevj = E_prev + j; E_tmp = E + j;
512,000 0 for(i = 0; i < n; i++) {
721,408,002 16,630,001 E_tmp[i] = E_prevj[i]+alpha*(E_prevj[i+1]...)
}
// Solve the ODEs
4,000 4,000 for(j=innerBlkRowStartIndx; j <= innerBlkRowEndIndx;j+=(m+3)){
for(i = 0; i <= n; i++) {
262,144,000 33,028,000 E_tmp[i] += -dt*(kk*E_tmp[i]*(E_tmp[i]-a)...) *R_tmp[i];
393,216,000 4,000 R_tmp[i] += dt*(ε+M1*R_tmp[i]/(E_tmp[i]+M2))*(...);
}
```

Scott B. Baden / CSE 260, UCSD / Fall '15

15
Fusing the loops

- Slows down the simulation by 20%
- # data references drops by 35%, total number of read misses drops by 48%
- What happened?

For a specified number of iterations, using supplied initial conditions repeat

for (j=1; j < m+1; j++) {
    for (i=1; i < n+1; i++) {
        // PDE SOLVER
        E[j,i] = E_p[j,i] + α*(E_p[j,i+1]+E_p[j,i-1]-4*E_p[j,i]+E_p[j+1,i]+E_p[j-1,i]);
        // ODE SOLVER
        E[j,i] += -dt*(kk*E[j,i]*(E[j,i]-a)*(E[j,i]-1)+E[j,i]*R[j,i]);
        R[j,i] += dt*(ε+M1* R[j,i]/(E[j,i]+M2))*(-R[j,i]-kk*E[j,i]*(E[j,i]-b-1));
    }
}
swap E_p and E
End repeat
Today’s lecture

• Stencil methods on the GPU
• Aliev Panfilov Method (A3)
• CUDA Implementation
Naïve CUDA Implementation

- All array references go through device memory

```c
#define E'[i,j] E_prev[j*(m+2) + i]
J = blockIdx.y*blockDim.y + threadIdx.y;
I = blockIdx.x*blockDim.x + threadIdx.x;
if ((I > 0) && (I <= n) && (J > 0) && (J <= m))
```

```c
for (j=1; j<= m+1; j++)
    for (i=1; i<= n+1; i++)
        E[j][i] = E'[j][i] + \alpha*(E'[j][i-1] + E'[j][i+1] +
                                  E'[j-1][i] + E'[j+1][i] - 4*E'[j][i]);
```
Using Shared Memory (Device Cap 3.5, 3.7 & 1.3)

\[ E^t[i,j] = E^{t-1}[i,j] + \alpha(E^{t-1}[i+1,j] + E^{t-1}[i-1,j] + E^{t-1}[i,j+1] + E^{t-1}[i,j-1] - 4E^{t-1}[i,j]) \]
Handling Ghost Cells

• Two approaches
  ♦ Assign 1 thread to each mesh cell including ghost cells
  ♦ Assign 1 thread to interior cells only
• Divide work between threads so each thread responsible for one ghost cell load
• Thread divergence, but cost differs
• For 16 × 16 tile size: $16 \times 4 - 4 = 60$ ghost cells

Ghost cells
Each thread loads one ghost cell
Thread uses a hash map to find its ghost cell assignment
Thread assignment

- We assign a thread to interior cells only
- 3 phases
  1. Fill interior
  2. Fill ghost cells – red circles correspond to active threads, orange to ghost cell data they copy into shared memory
  3. Compute – uses the same threads as step 1
CUDA Code to load shared memory

```c
__shared__ float block[DIM_Y + 2][DIM_X + 2];
int idx = threadIdx.x, idy = threadIdx.y; //local indices
//global indices
int x = blockIdx.x * (DIM_X) + idx;
int y = blockIdx.y * (DIM_Y) + idy;
idy++; idy++; // Why do we increment?
unsigned int index = y * N + x;

//interior points
float center = E_prev[index]; // Where is center stored?
block[idy][idx] = center;

__syncthreads();
```
Copying the ghost cells

1 thread for each interior cell

if (idy == 1 && y > 0 )
    block[0][idx] = E_prev[index - N];
else if(idy == DIM_Y && y < N-1)
    block[DIM_Y+1][idx] = E_prev[index + N];
if ( idx==1 && x > 0 )
    block[idy][0] = E_prev[index - 1];
else if( idx== DIM_X && x < N-1 )
    block[idy][DIM_X +1] = E_prev[index + 1];
__syncthreads();
The stencil computation and ODE

float r = R[index];

float e = center + α*(block[idy][idx-1] + block[idy][idx+1] + block[idy-1][idx] + block[idy+1][idx] - 4*center);

\[ e = e - dt*(kk*e*(e-a)*(e-1) + e*r); \]

E[index] = e;

R[index] = r + dt*(ε+ M1 * r / (e + M2)) * (-r - kk * e * (e - b - 1));
Sliding rows scheme

- Instead of using a 2D block of memory, we only keep 1 row in shared memory, the rest in registers.
- Sliding rows with 1D thread blocks reduces global memory accesses.

Top row $\leftarrow$ Current row,
Current row $\leftarrow$ Bottom row
Bottom row $\leftarrow$ read new row from global memory
Inter-thread communication

- Historically, GPUs supported *intra-block* communication via *shared memory*, and *inter-block* via *global memory*.
- Shared memory and registers do not persist across kernel invocations.
- Kepler (CC 3.0+) supports high speed atomics in global memory, that are suitable for inner loops (e.g., summation).
- New instructions support direct data exchange between threads in the same warp.
- A thread can read values stored in a register from a thread within the same warp.
  - Shift
- See
  - http://blog.csdn.net/u010060391/article/details/21558557