Peak Performance for an Application in CUDA

Nicolás Wolovick[1]

Fa.M.A.F., Universidad Nacional de Córdoba, Argentina

May 4, 2010

Fa.M.A.F. - U.N.C.

Outline

The Numeric Problem

Serial Codes

Our aim

CUDA Codes

Conclusions

The Numeric Problem

Serial Codes

Our aim

CUDA Codes

Conclusions

Heat Conduction in a Plate

Also known as Laplace Differential Equation in two dimensions.

$$\nabla^2 T = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

It can be solved iteratively using:

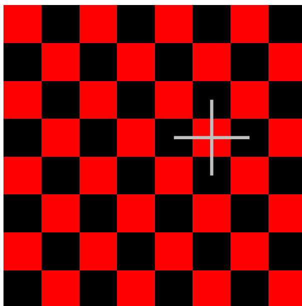
- Jacobi
- Gauss-Seidel
- Red-Black Gauss-Seidel

Red-Black Gauss-Seidel

A highly parallel, memory efficient method

Description

1st pass: update red cells from n to $n+1$ using blacks in n.
2nd pass: update black cells from n to $n+1$ using reds in $n+1$.



Red-Black Gauss-Seidel Iteration

Red

For each **red** (i, j), that is $(i + j) \mod 2 = 0$ do:

$$M(i,j) \doteq \frac{M(i-1,j) + M(i,j-1) + M(i+1,j) + M(i,j+1)}{4}$$

Black

For each **black** (i, j), that is $(i + j) \mod 2 = 1$ do:

$$M(i,j) \doteq \frac{M(i-1,j) + M(i,j-1) + M(i+1,j) + M(i,j+1)}{4}$$

(stencil average)

Borders are not updated since they are our *contour conditions*.

The Numeric Problem

Serial Codes

Our aim

CUDA Codes

Conclusions

Serial_Checkboard

Remarks:

- Massive use of `asserts`
  ...they can be disabled with `-DNDEBUG` define.
- Border of fire of cuadratic temperature.
  Maximum in the center.
- Write simple `PPMs` image files as Pre/Post state.
- Measure walltime with microsecond precision.
- Print stats: GFlops and GBps.

Serial_Checkboard, results

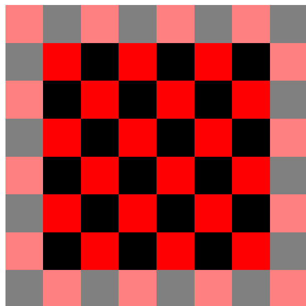
```
gcc -std=gnu99 -O3 -Wall -DNDEBUG -DMAX_ITERATIONS=1000   -c -o
gcc -std=gnu99 -O3 -Wall -DNDEBUG -DMAX_ITERATIONS=1000   -c -o
gcc heat.o ../../common/common.o -o heat -std=gnu99 -O3 -Wall -D
./heat
Secs: 12.777368
GBps: 1.641302
GFlops: 0.328260
```

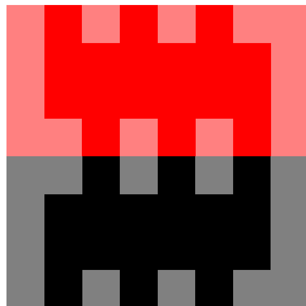
Disclaimer

Our GBps, our GFlops, perhaps nothing to do with the real x86 architecture, but it serves for comparison.

Serial_Checkboard_Packed

$$(i,j) \rightarrow \left(\frac{(i+j) \mod 2 \times N + i}{2}, j \right)$$





Serial_Checkboard_Packed, hoping less cache misses

With this transformation there are three reads in a row.

Thanks to cache *spatial locality* they implement prefetching bringing a whole cache line.

Serial_Checkboard_Packed, hoping less cache misses

With this transformation there are three reads in a row.

Thanks to cache *spatial locality* they implement prefetching bringing a whole cache line.

Running it

```
./heat
Secs: 24.516243
GBps: 0.855413
GFlops: 0.171083
```

Not so useful.

Serial_Checkboard_Packed, less cache misses?

Using tools/perf/perf to quantify the executions.

```
Serial_Checkboard$ perf stat ./heat
Secs: 13.877461
GBps: 1.511193
GFlops: 0.302239

 Performance counter stats for './heat':

    29989928783   cycles
    17154205308   instructions
      133880188   cache-references
        3208543   cache-misses

   14.008524450   seconds time elapsed
```

```
Serial_Checkboard_Packed$ perf stat ./heat
Secs: 22.524494
GBps: 0.931054
GFlops: 0.186211

 Performance counter stats for './heat':

    48381072692   cycles
    26476442265   instructions
      138237703   cache-references
        4295066   cache-misses

   22.827983029   seconds time elapsed
```

It seems

- Double instruction count.
- No cache hits payoff.

The Numeric Problem

Serial Codes

Our aim

CUDA Codes

Conclusions

Peak performance for Red-Black Gauss-Seidel

- Memory-bandwidth limited problem.
- Measure memory bandwith for the architecture.
- Use the knowledge of the underlying architecture to reach such value.

We are going to measure the BW using SAXPY.

$$\vec{y} = \alpha \vec{x} + \vec{y}$$

SAXPY/Serial

Architecture

Intel Core 2 Duo E8400, 2GB DDR3 1333MHz
(*January 2008*)

SAXPY measurment

For a vector of 2^{24} `floats`.

```
Secs: 0.030218
GBps: 6.662472
GFlops: 1.110412
```

Wikipedia informs 10.67 GBps.

SAXPY/CUDA

Architecture

NVIDIA GTX 280 (GT200 arch), 1GB GDDR3, 30*8=240 cores (*June 2008*)

SAXPY measurment

For a vector of 2^{24} `floats`, blocks of 512 threads.

```
Secs: 0.001747
GBps: 115.241323
GFlops: 19.206887
```

Wikipedia informs 141.70 GBps.

Speedup

For a memory bandwidth intensive application:

17x

We aim to peak in our app.

115GBps

The Numeric Problem

Serial Codes

Our aim

CUDA Codes

Conclusions

CUDA_Checkboard

code inspection

CUDA_Checkboard

code inspection

We emphasize

- All red(black) 512×1024 threads are divided in
  - A block of 16×16 threads.
  - A grid of 32×64 blocks.
- First map thread position to matrix element.
- Then compute the average.

We are (still) surprised with

- Extra fine granularity for concurrency.
- Pure C plus minor extensions.

CUDA_Checkboard, results

Running it

```
Secs: 0.403626
GBps: 51.957803
GFlops: 10.391561
```

Speedup

$$\frac{Serial_Checkboard}{CUDA_Checkboard} = \frac{12.777368}{0.403626} \geq 31x$$

Correctness

```
$ diff Serial_Checkboard/image_after.ppm CUDA_Checkboard/image_after.ppm
Binary files Serial_Checkboard/... and CUDA_Checkboard/... differ
```

Not so. . .

CUDA_Checkboard_Packed

code inspection

CUDA_Checkboard_Packed

code inspection

Remarks

- Avoid branches using arithmetic.
- The only `if (...) { ...}` is for border detection.
- Divergent branches makes warps slow.

We hope for the best.

CUDA_Checkboard_Packed, results

Running it

```
Secs: 0.266050
GBps: 78.825484
GFlops: 15.765097
```

Speedup

$$\frac{\textit{Serial_Checkboard}}{\textit{CUDA_Checkboard}} = \frac{12.777368}{0.266050} \geq 48x$$

Do some performance measurments to find out why.

CUDA_Checkboard{_Packed}, profiling

CUDA_Checkboard

```
# CUDA_PROFILE_LOG_VERSION 1.5
# CUDA_DEVICE 0 GeForce GTX 280
# TIMESTAMPFACTOR fe5c7d8438c7794
method,gputime,cputime,registerPerThread,occupancy,gld_32b,gld_64b,gld_128b,divergent_branch
method=[ memcpyHtoD ] gputime=[ 1535.264 ] cputime=[ 1865.000 ]
method=[ _Z8heat_redPf ] gputime=[ 200.032 ] cputime=[ 215.000 ] registerPerThread=[ 6 ]
    occupancy=[ 1.000 ] gld_32b=[ 3162 ] gld_64b=[ 22914 ] gld_128b=[ 3274 ] divergent_branch=[ 28 ]
method=[ _Z10heat_blackPf ] gputime=[ 203.104 ] cputime=[ 211.000 ] registerPerThread=[ 6 ]
    occupancy=[ 1.000 ] gld_32b=[ 3178 ] gld_64b=[ 22915 ] gld_128b=[ 3273 ] divergent_branch=[ 20 ]
```

CUDA_Checkboard_Packed

```
# CUDA_PROFILE_LOG_VERSION 1.5
# CUDA_DEVICE 0 GeForce GTX 280
# TIMESTAMPFACTOR fe5c7d92dd8961c
method,gputime,cputime,registerPerThread,occupancy,gld_32b,gld_64b,gld_128b,divergent_branch
method=[ memcpyHtoD ] gputime=[ 381.472 ] cputime=[ 1111.000 ]
method=[ memcpyHtoD ] gputime=[ 381.440 ] cputime=[ 1030.000 ]
method=[ _Z8heat_redPfS_ ] gputime=[ 134.752 ] cputime=[ 149.000 ] registerPerThread=[ 6 ]
    occupancy=[ 1.000 ] gld_32b=[ 3162 ] gld_64b=[ 13113 ] gld_128b=[ 3274 ] divergent_branch=[ 29 ]
method=[ _Z10heat_blackPfS_ ] gputime=[ 136.992 ] cputime=[ 146.000 ] registerPerThread=[ 6 ]
    occupancy=[ 1.000 ] gld_32b=[ 3178 ] gld_64b=[ 13114 ] gld_128b=[ 3273 ] divergent_branch=[ 20 ]
```

Thanks to hardware performance counters.

The Numeric Problem

Serial Codes

Our aim

CUDA Codes

Conclusions

Lessons learnt

- Avoid branches using arithmetic boolean-encoding tricks:
  `i+(2*(j%2)-1)`
- Avoid (non) optimizations that hinder readability:
  `i!=(N-1)/2*(j&1))` identical to `i!=(N-1)/2*(j%2))`
  `i>>=1` identical to `i/=2`.
- `nvcc` is a modern compiler.
- Disconnect profiling: `export CUDA_PROFILE=0`.
- Compile to `PTX` and briefly inspect the code.
  We discovered `0.25f*(...)` is way different to `0.25*(...)`
- Play with block size and grid size in powers of two.
- Use all the (scarce) runtime error detection mechanisms:
  `CUDA_SAFE_CALL(...)`, `CUT_CHECK_ERROR(...)`.
- Profile, measure, experiment, propose explaining hypothesis, etc.

Conclusions

We obtained 67% of peak memory bandwidth, and a considerable speedup: 57x.
The code is still readable for humans.

Conclusions

We obtained 67% of peak memory bandwidth, and a considerable speedup: 57x.
The code is still readable for humans.

Thanks!

Tomorrow we are going to deal with summarizations of the grid.
We need to know *when to stop*.

The Attic

Spare time?

The Attic

Spare time? Ok, let's show the little weirdos of the attic.

The Attic

Spare time? Ok, let's show the little weirdos of the attic.

What if we just program the **trivial** Jacobi iteration?

- Double matrix (swap old&new each turn).
- Fully parallel.

The Attic

Spare time? Ok, let's show the little weirdos of the attic.

What if we just program the **trivial** Jacobi iteration?

- Double matrix (swap old&new each turn).
- Fully parallel.

`Serial_Trivial`

```
Secs: 25.292405
GBps: 0.829163
GFlops: 0.165833
```

Too bad for the serial case.

The Attic, the freak: CUDA_Trivial

code inspection

The Attic, the freak: `CUDA_Trivial`

code inspection

Running it

```
Secs: 0.218067
GBps: 96.170076
GFlops: 19.234015
```

Speedup

$$\frac{Serial_Checkboard}{CUDA_Trivial} = \frac{12.777368}{0.218076} \geq 58x$$

Amazing: zero effort, big gains, near peak bandwidth.

The Attic, the freak, the counters

CUDA_Checkboard_Packed

```
# CUDA_PROFILE_LOG_VERSION 1.5
# CUDA_DEVICE 0 GeForce GTX 280
# TIMESTAMPFACTOR fe5c7d92dd8961c
method,gputime,cputime,registerPerThread,occupancy,gld_32b,gld_64b,gld_128b,divergent_branch
method=[ memcpyHtoD ] gputime=[ 381.472 ] cputime=[ 1111.000 ]
method=[ memcpyHtoD ] gputime=[ 381.440 ] cputime=[ 1030.000 ]
method=[ _Z8heat_redPfS_ ] gputime=[ 134.752 ] cputime=[ 149.000 ] registerPerThread=[ 6 ]
    occupancy=[ 1.000 ] gld_32b=[ 3162 ] gld_64b=[ 13113 ] gld_128b=[ 3274 ] divergent_branch=[ 29 ]
method=[ _Z10heat_blackPfS_ ] gputime=[ 136.992 ] cputime=[ 146.000 ] registerPerThread=[ 6 ]
    occupancy=[ 1.000 ] gld_32b=[ 3178 ] gld_64b=[ 13114 ] gld_128b=[ 3273 ] divergent_branch=[ 20 ]
```

CUDA_Trivial

```
# CUDA_PROFILE_LOG_VERSION 1.5
# CUDA_DEVICE 0 GeForce GTX 280
# TIMESTAMPFACTOR fe5c7db01f0ad18
method,gputime,cputime,registerPerThread,occupancy,gld_32b,gld_64b,gld_128b,divergent_branch
method=[ memcpyHtoD ] gputime=[ 1535.424 ] cputime=[ 1872.000 ]
method=[ memcpyHtoD ] gputime=[ 1548.256 ] cputime=[ 1934.000 ]
method=[ _Z4heatPfS_ ] gputime=[ 220.224 ] cputime=[ 238.000 ] registerPerThread=[ 6 ]
    occupancy=[ 1.000 ] gld_32b=[ 6340 ] gld_64b=[ 19641 ] gld_128b=[ 6547 ] divergent_branch=[ 44 ]
```

There is no apparent reason, everything is worse.

The Attic, the freak, the reasons

Reading from global memory is a hary business:

- Alignment
- How warps access the memory in each instruction

We hope a different transformation (paralelogram?) can render better results.

The Attic, the freak using shared: `CUDA_Trivial_Shared`

code inspection

The Attic, the freak using shared: `CUDA_Trivial_Shared`

code inspection

Running it

```
Secs: 0.242293
GBps: 86.554378
GFlops: 17.310876
```

Profiling it

```
method,gputime,cputime,registerPerThread,occupancy,gld_32b,gld_64b,gld_128b,divergent_branch
method=[ memcpyHtoD ] gputime=[ 1535.200 ] cputime=[ 1872.000 ]
method=[ memcpyHtoD ] gputime=[ 1544.320 ] cputime=[ 1932.000 ]
method=[ _Z4heatPfS_ ] gputime=[ 244.096 ] cputime=[ 262.000 ] registerPerThread=[ 7 ]
    occupancy=[ 1.000 ] gld_32b=[ 4441 ] gld_64b=[ 4680 ] gld_128b=[ 3989 ] divergent_branch=[ 1530 ]
```

Each cell is read four times. Although in shared (ten to a hundred times faster), the more miss-aligned reads, the logic, and divergent branches make it slower.

Finally, two experiments

- Completely simetrical problem, swap i, j and see.
- $TILE - 2$ seems awkward to memory alignmen, what if just $TILE$?

See you tomorrow.

Questions?