Peak Performance for an Application in CUDA

Nicolás Wolovick[1]

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

May 5, 2010

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

Outline

Stopping condition

Up to this point there were a fixed number of iterations.

Stop when there is not enough change in the iterations:

$$(\max i,j : 0 \leq i,j < N : |M'(i,j) - M(i,j)|) < \epsilon$$

- The computation is both local (update) and global (max reduction).
- Example of other reduction problem: configuration energy. Big sums.

We are going to modify the fastest codes:

- `Serial_Checkboard`
- `CUDA_Checkboard_Packed`

Serial

code inspection

Serial

code inspection

Running it

```
gcc -std=gnu99 -O3 -Wall -DNDEBUG -DMAX_ITERATIONS=3000   -c -o
gcc -std=gnu99 -O3 -Wall -DNDEBUG -DMAX_ITERATIONS=3000   -c -o
gcc heat.o ../../common/common.o -o heat -std=gnu99 -O3 -Wall -D
./heat
Iters: 1205
Secs: 19.584123
GBps: 1.548439
GFlops: 0.451628
```

Overhead

$$\frac{1.641302\ GBps}{1.548439\ GBps} = 6\%$$

General strategies for paralelization

- Computation of the maximum
  - One global variable, every thread hitting the maximum.
  - Distributed (butterfly) computation of the maximum.
- Predicate transformation
  - One global variable, every thread hitting the boolean.
  - Distributed (butterfly) computation of the boolean.

Equivalent forms

$$(\max i,j : 0 \leq i,j < N : |M'(i,j) - M(i,j)|) < \epsilon$$

$$\equiv$$

$$(\forall i,j : 0 \leq i,j < N : |M'(i,j) - M(i,j)| < \epsilon)$$

Maximum vs. Predicate computation

Maximum gives a strictly increasing value.

$$(\max i,j : 0 \leq i,j < N : |M'(i,j) - M(i,j)|) < \epsilon$$

$$0.0, 0.271, 0.85, 0.90, 1.02, \ldots, 1.19997, 1.19998$$

$\forall$ computation gives a strictly decreasing ($\Rightarrow$ order) function.

$$(\forall i,j : 0 \leq i,j < N : |M'(i,j) - M(i,j)| < \epsilon)$$

$$\mathit{true}, \mathit{true}, \ldots, \mathit{true}, \mathit{false}, \ldots, \mathit{false}$$

Mhhh, much more suitable for tricks.

Convergence condition

Serial Codes

Paralelization

CUDA Codes

Conclusions

CUDA_Global_Diff

code inspection

CUDA_Global_Diff

code inspection

Forecasting

- Correctness
- Performance

CUDA_Global_Diff

code inspection

Forecasting

- Correctness
- Performance

Running it

```
/opt/cuda/bin/nvcc heat.o ../../common/common.o -o heat -arch=sm_13 -O3 -I/home/nicolasw/NVIDIA_CUDA_SDK/c
./heat
0 23.48595
100 7.10007
200 2.80053
300 3.19948
400 3.01183
500 2.69680
600 3.03030
700 2.19561
2700 1.53413
2800 1.44889
2900 1.37213
Iters: 3000
Secs: 6.139613
GBps: 12.296780
GFlops: 3.074195
```

CUDA_Global_Diff, Correctness

Race condition? Missed updates? Unstable predicate? Non global correctness? Whatever, either if you are Formal Methods or Concurrency or I-just-program guy.

The problematic line

```
*diff = max(*diff, fabs(cnew-cold));
```

PTX

```
ld.global.f32   %f10, [%rd10+0];        // id:63
ld.global.f32   %f11, [%rd9+0];         // id:64
sub.f32         %f12, %f9, %f11;        //
abs.f32         %f13, %f12;             //
max.f32         %f14, %f10, %f13;       //
st.global.f32   [%rd10+0], %f14;        // id:65
```

Solutions

`atomicMax()`,

CUDA_Global_Diff, Correctness

Race condition? Missed updates? Unstable predicate? Non global correctness? Whatever, either if you are Formal Methods or Concurrency or I-just-program guy.

The problematic line

```
*diff = max(*diff, fabs(cnew-cold));
```

PTX

```
ld.global.f32    %f10, [%rd10+0];        // id:63
ld.global.f32    %f11, [%rd9+0];         // id:64
sub.f32          %f12, %f9, %f11;        //
abs.f32          %f13, %f12;             //
max.f32          %f14, %f10, %f13;       //
st.global.f32    [%rd10+0], %f14;        // id:65
```

Solutions

`atomicMax()`, not so, for Compute Capability 1.3, there is no `float` version.

`CUDA_Global_Diff`, Performance

Down on its knees.

$$\frac{12.296780\ GBps}{78.825484\ GBps} = 15\%$$

`atomicMax()` for `CUDA 3.0` will only solve correctness, not performance.

We have to go to a more distributed computation.

CUDA_Distributed_Diff

code inspection

CUDA_Distributed_Diff

code inspection

General Description

- Each block computes the maximum of $16 \times 16 = 256$ subgrid.
- This is computed using a log-stage algorithm.
- Write those $32 \times 64 = 2048$ floats to global memory.
- Those 2048 values are reduced in the host.

Remarks

- Real use of `__shared__`.
- Inter block barrier sincronization.
- Two coordinate system (i, j) vs. (tid, bid).

There are good support for concurrency control within blocks.

CUDA_Distributed_Diff, results

Running it

```
./heat
0 37.49995
100 0.24138
200 0.12075
300 0.08048
1100 0.02190
1200 0.02007
Iters: 1206
Secs: 0.556048
GBps: 54.581590
GFlops: 13.645397
```

Remarks

- Converges!
- It is quite fast: $\frac{54.581590 GBps}{78.825484 GBps} = 69\%$.
- One more iteration wrt the serial version (IEEE754 glitches?).

CUDA_Distributed_Predicate

code inspection

CUDA_Distributed_Predicate

code inspection

Code snippet

```
/* Block flag reset */
if (tid==0)
        blockAll_LE_Epsilon = true;
__syncthreads();
/* Warp voting for all below Epsilon */
bool warpAll_LE_Epsilon = __all(fabs(cnew-cold)<=EPSILON);
/* Each warp first thread can reset the block value */
if (tid%32==0 && !warpAll_LE_Epsilon)
        blockAll_LE_Epsilon = false;
__syncthreads();
all[bid] = blockAll_LE_Epsilon;
```

Use and abuse of the boolean properties.

CUDA_Distributed_Predicate, results

Running it

```
Iters: 1206
Secs: 0.523146
GBps: 58.014366
GFlops: 14.503592
```

Slight performance increase

$$\frac{58.014366 GBps}{54.581590 GBps} = 6\%$$

Remarks

- Instead of bool *, we used int * for the global array.
- `sizeof(bool)=1`, and this introduces contemption, `sizeof(int)=4` improves the situation.
- Thanks to Charly Bederián for pointing this out.

CUDA_Global_Predicate, motivation

Our last success, was the simplest solution.
We play the naïve once more.

- One global boolean variable.
- Every thread hitting there.

code inspection

CUDA_Global_Predicate, motivation

Our last success, was the simplest solution.
We play the naïve once more.

- One global boolean variable.
- Every thread hitting there.

code inspection

Running it

```
Iters: 1206
Secs: 0.405145
GBps: 74.911411
GFlops: 18.727853
```

Extra nice again, simple and extremely good performing.

Lessons learnt

- Every one hitting the same memory cell is bad.
- Concurrency is a difficult problem in CUDA.
- Ye Olde butterfly reduction algorithms are in fashion again.

In general, all tipical concurrency problems are amplified in a context involving millons of threads.

Conclusions

We slightly penalized our code for convergence computation:

$$\frac{78.825484 \;\; GBps}{74.911411 \;\; GBps} = 5\%$$

In case of really needing a sum or a max we have an impact of:

$$\frac{78.825484 \;\; GBps}{54.581590 \;\; GBps} = 44\%$$

A common problem for Massively Parallel Processors.

Conclusions

We slightly penalized our code for convergence computation:

$$\frac{78.825484 \ \ GBps}{74.911411 \ \ GBps} = 5\%$$

In case of really needing a sum or a max we have an impact of:

$$\frac{78.825484 \ \ GBps}{54.581590 \ \ GBps} = 44\%$$

A common problem for Massively Parallel Processors.

Thanks!

GPGPU Computing Group Fa.M.A.F. – U.N.C.

Funding

- Professor Partnership for Oscar Reula, Sep 2008.
- Two NVidia GTX280, Sep 2008.

Production

- 1400x for Post Newtonian equations.
- 100x-200x for Potts Model.
- 21x for Trotter-Suzuki.
- Currently digging down problems with Ricci Flow.
- Education: two mini-courses, three talks, incorporating GPGPU computing to Concurrency and HPC courses.
- Colaboration with UMD's CSCAMM.
- One final work for a CS undergraduate (Peak Performance . . .).

GPGPU Computing Group Fa.M.A.F. – U.N.C.

Please visit us at:

`http://cs.famaf.unc.edu.ar/ nicolasw/GPGPU/`