Determinant Computation on the GPU using the Condensation Method

Sardar Anisul Haque
Marc Moreno Maza

Ontario Research Centre for Computer Algebra
University of Western Ontario, London, Ontario

AMMCS 2011, Waterloo, July 25, 2011

Plan

Dodgson's condensation Algorithm

- Example of a condensation step:

$$\begin{vmatrix} 0 & -1 & 2 \\ -1 & -5 & 8 \\ 1 & 1 & -4 \end{vmatrix}$$

$=>$

$$\begin{vmatrix} \begin{vmatrix} 0 & -1 \\ -1 & -5 \end{vmatrix} & \begin{vmatrix} -1 & 2 \\ -5 & 8 \end{vmatrix} \\ \begin{vmatrix} -1 & -5 \\ 1 & 1 \end{vmatrix} & \begin{vmatrix} -5 & 8 \\ 1 & -4 \end{vmatrix} \end{vmatrix}$$

- **Reference:**
  C. L. Dodgson, *Condensation of Determinants*, Proceedings of the Royal Society of London, 15(1866), 150-155.

Dodgson's condensation Algorithm (cont.)

- Condensation step (cont.)

$$\begin{vmatrix} 0 & -1 & 2 \\ -1 & {\color{red}-5} & 8 \\ 1 & 1 & -4 \end{vmatrix}$$

$=>$

$$\begin{vmatrix} -1 & 2 \\ 4 & 12 \end{vmatrix}$$

$= -20$

- And the determinant is: $-20/{\color{red}-5} = 4$.

Salem and Said's condensation Algorithm

- Condensation step with the same example:

$$\begin{vmatrix} 0 & \color{red}{-1} & 2 \\ -1 & -5 & 8 \\ 1 & 1 & -4 \end{vmatrix}$$

$=>$

$$\begin{vmatrix} \begin{vmatrix} 0 & \color{red}{-1} \\ -1 & -5 \end{vmatrix} & \begin{vmatrix} \color{red}{-1} & 2 \\ -5 & 8 \end{vmatrix} \\ \\ \begin{vmatrix} 0 & \color{red}{-1} \\ 1 & 1 \end{vmatrix} & \begin{vmatrix} \color{red}{-1} & 2 \\ 1 & -4 \end{vmatrix} \end{vmatrix}$$

- A formula is needed before concluding (see next slide).
- **Reference:**
  Abdelmalek Salem, and Kouachi Said, *Condensation of Determinants*, http://arxiv.org/abs/0712.0822.

Salem and Said's condensation Algorithm (cont.)

- The input of a condensation step is a matrix
$$A = (a_{i,j} \mid 0 \leq i,j \leq n-1)$$
of order $n > 2$.
- It produces a matrix $B = (b_{i,j} \mid 0 \leq i,j \leq n-1)$ of order $n-1$ such that
$$b_{i,j} = \begin{vmatrix} a_{0,\ell} & a_{0,j+1} \\ a_{i+1,\ell} & a_{i+1,j+1} \end{vmatrix}$$
for $j \geq \ell$ and by $b_{i,j} = -a_{i+1,j}a_{0,\ell}$ for $j < \ell$.
- The key relation between A and B is the following:
$$\det(A) = \det(B)/(a_{0,\ell})^{n-2}$$

Salem and Said's condensation Algorithm (cont.)

- Returning to our example, we obtain:

$$\begin{vmatrix} 0 & -1 & 2 \\ -1 & -5 & 8 \\ 1 & 1 & -4 \end{vmatrix}$$

$=>$

$$\begin{vmatrix} -1 & 2 \\ 1 & 2 \end{vmatrix}$$

$=> -4$

- So the determinant is: $-4/(-1)^{3-2} = 4$.

Complexity estimates for Salem and Said's Algorithm

- For the usual RAM model, In the worst case, the work is $n^3 - 3/2n^2 + 1/2n - 3$ coefficient operations.
- Asymptotically, on (Z, L) ideal cache, the number of cache misses is in the order of

$$\frac{(n - Z)\left(n^2 - n + Z^2 - Z + Zn + 1 + 4L\right)}{L}$$

- Hence, the ratio between the two is L, similarly to Gaussian Elimination.
- However, the condensation method is more **data-oblivious** which is good for the hardware scheduling of a GPU.

Plan

Data mapping

1. Each condensation step is performed by one kernel call. No data copied back to the host until $n = 2$.
2. At each condensation step, the input A and output B are stored in global memory. **Shared memory is used** for efficiency issues.
3. Salem and Said's Algorithm suggest to store A and B in **column major layout**.
4. We use a **1-D grid of 1-D thread blocks**.
5. With T threads per block and t elements of B written per thread, $\lceil (n-1)^2/(Tt) \rceil$ blocks are required. For $t = 4$ and $n > 200$ this leads to about $10,000$ threads in flight.
6. The j-th thread in the i-th block computes-and-writes $B[Ttj + it, Ttj + it + 1, \ldots Ttj + it + t - 1]$.

Data mapping

1. Each condensation step is performed by one kernel call. No data copied back to the host until $n = 2$.
2. At each condensation step, the input A and output B are stored in global memory. **Shared memory is used** for efficiency issues.
3. Salem and Said's Algorithm suggest to store A and B in **column major layout**.
4. We use a **1-D grid of 1-D thread blocks**.
5. With T threads per block and t elements of B written per thread, $\lceil (n-1)^2/(Tt) \rceil$ blocks are required. For $t = 4$ and $n > 200$ this leads to about $10,000$ threads in flight.
6. The j-th thread in the i-th block computes-and-writes $B[Ttj + it, Ttj + it + 1, \ldots Ttj + it + t - 1]$.

Data mapping

1. Each condensation step is performed by one kernel call. No data copied back to the host until $n = 2$.
2. At each condensation step, the input A and output B are stored in global memory. **Shared memory is used** for efficiency issues.
3. Salem and Said's Algorithm suggest to store A and B in **column major layout**.
4. We use a **1-D grid of 1-D thread blocks**.
5. With T threads per block and t elements of B written per thread, $\lceil (n-1)^2/(Tt) \rceil$ blocks are required. For $t = 4$ and $n > 200$ this leads to about $10,000$ threads in flight.
6. The j-th thread in the i-th block computes-and-writes $B[Ttj + it, Ttj + it + 1, \ldots Ttj + it + t - 1]$.

Finding the ℓ-th column: finite field case

- A condensation step produces a matrix $B = (b_{i,j} \mid 0 \le i, j \le n-1)$ of order $n-1$ such that

$$b_{i,j} = \begin{vmatrix} a_{0,\ell} & a_{0,j+1} \\ a_{i+1,\ell} & a_{i+1,j+1} \end{vmatrix}$$

  for $j \ge \ell$ and by $b_{i,j} = -a_{i+1,j} a_{0,\ell}$ for $j < \ell$.
- Recall that we have

$$\det(A) = \det(B) / (a_{0,\ell})^{n-2}$$

- The above formula requires $a_{0,\ell}$ to be the first non-zero on the first row: we call it the pivot. It is computed by one kernel call.
- The successive pivots are in the global memory of GPU and used to compute the determinant of the original matrix.

Finding the ℓ-th column: finite field case

- A condensation step produces a matrix $B = (b_{i,j} \mid 0 \le i,j \le n-1)$ of order $n-1$ such that

$$b_{i,j} = \begin{vmatrix} a_{0,\ell} & a_{0,j+1} \\ a_{i+1,\ell} & a_{i+1,j+1} \end{vmatrix}$$

  for $j \ge \ell$ and by $b_{i,j} = -a_{i+1,j}a_{0,\ell}$ for $j < \ell$.
- Recall that we have

$$\det(A) = \det(B)/(a_{0,\ell})^{n-2}$$

- The above formula requires $a_{0,\ell}$ to be the first non-zero on the first row: we call it the pivot. It is computed by one kernel call.
- The successive pivots are in the global memory of GPU and used to compute the determinant of the original matrix.

Finding the ℓ-th column: floating point number Case

- On the first row, we choose the element p whose absolute value is the closest to 1: we call it the pivot.
- Then we divide each element of the first row by p and we have

$$\det(A) = \det(B) * p$$

- The successive pivots are stored in the GPU global memory.
- After all condensation steps are completed, the pivots are multiplied together so as to avoid overflow/underflow, if possible:

Step 1 $L_1 := \{p \in \mathrm{Pivots} \mid -1 \leq p \leq 1\}$;
$L_2 := \{p \in \mathrm{Pivots} \mid p \notin L_1\}$;
$m := 1$;

Step 2 While L_1 and L_2 not empty do $m := m\ \mathrm{pop}(L_1)\ \mathrm{pop}(L_1)$

Step 3 Finish wih the non-empty stack, if any.

Finding the ℓ-th column: floating point number Case

- On the first row, we choose the element p whose absolute value is the closest to 1: we call it the pivot.
- Then we divide each element of the first row by p and we have

$$\det(A) = \det(B) * p$$

- The successive pivots are stored in the GPU global memory.
- After all condensation steps are completed, the pivots are multiplied together so as to avoid overflow/underflow, if possible:

Step 1 $L_1 := \{p \in \mathrm{Pivots} \mid -1 \leq p \leq 1\}$;
$L_2 := \{p \in \mathrm{Pivots} \mid p \notin L_1\}$;
$m := 1$;

Step 2 While L_1 and L_2 not empty do $m := m \ \mathrm{pop}(L_1)\ \mathrm{pop}(L_1)$

Step 3 Finish wih the non-empty stack, if any.

Finding the ℓ-th column: floating point number Case

- On the first row, we choose the element p whose absolute value is the closest to 1: we call it the pivot.
- Then we divide each element of the first row by p and we have

$$\det(A) = \det(B) * p$$

- The successive pivots are stored in the GPU global memory.
- After all condensation steps are completed, the pivots are multiplied together so as to avoid overflow/underflow, if possible:

Step 1 $L_1 := \{p \in \mathrm{Pivots} \mid -1 \leq p \leq 1\}$;
$L_2 := \{p \in \mathrm{Pivots} \mid p \notin L_1\}$;
$m := 1$;

Step 2 While L_1 and L_2 not empty do $m := m \,\mathrm{pop}(L_1)\, \mathrm{pop}(L_1)$

Step 3 Finish wih the non-empty stack, if any.

Plan

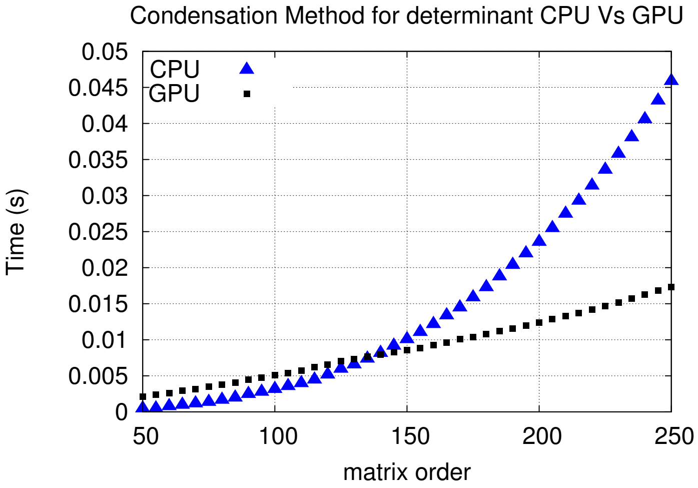
Experimental setup

- The order of our test matrices varies from 10 to 4000.
- We conduct all our experiments on a GPU NVIDIA Tesla 2050 C.
- Our GPU code is written using CUDA.
- Our CPU is *intel core 2 processor Q6600*. It has L2 cache of 8MB and the CPU frequency is 2.40 GHz.
- **Reference:** NVIDIA developer zone, http://developer.nvidia.com.

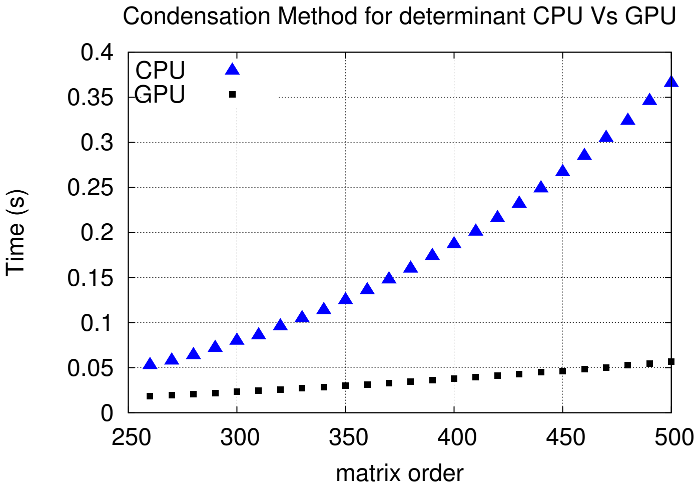
Effective memory bandwidth

- We use *effective memory bandwidth* to evaluate our GPU code.
- The effective memory bandwidth (measured in GB/seconds) of a kernel run is amount of data traversed in the global memory of GPU during the kernel run divided by the running time of the kernel.
- It is compared against a simple CUDA code, called *copy kernel*, that just performs one copy memory from one place to other place in the global area of GPU.
- **Reference:**
  Greg Ruetsch, and Paulius Micikevicius, *Optimizing Matrix Transpose in CUDA*, NVIDIA Corporation, 2009.

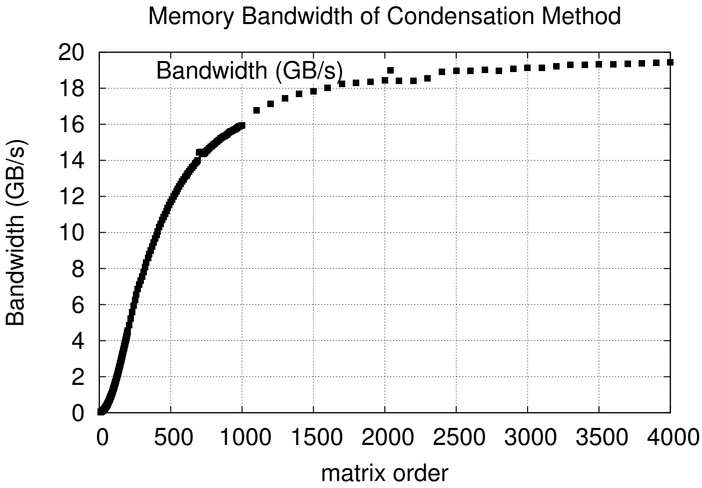
CPU Vs GPU


Condensation Method for determinant CPU Vs GPU
CPU
GPU
Time (s)
matrix order
0.05
0.045
0.04
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
50
100
150
200
250

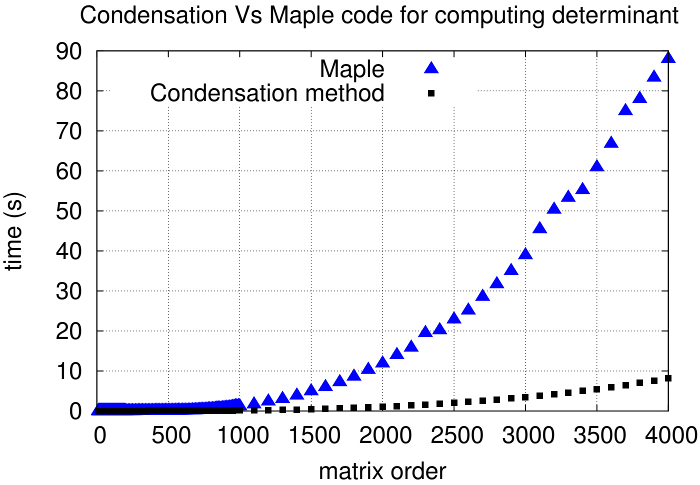
CPU Vs GPU (cont.)



Effective Memory Bandwidth (cont.)



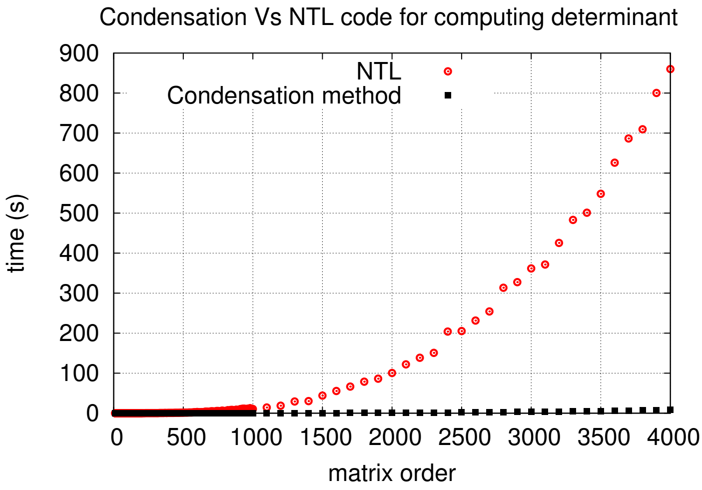
Finite Filed Case 1



Reference:

Maple: http://www.maplesoft.com.

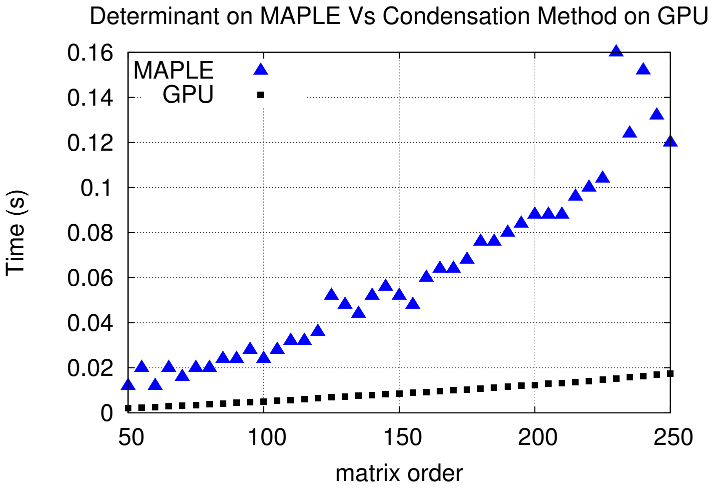
Finite Filed Case 2



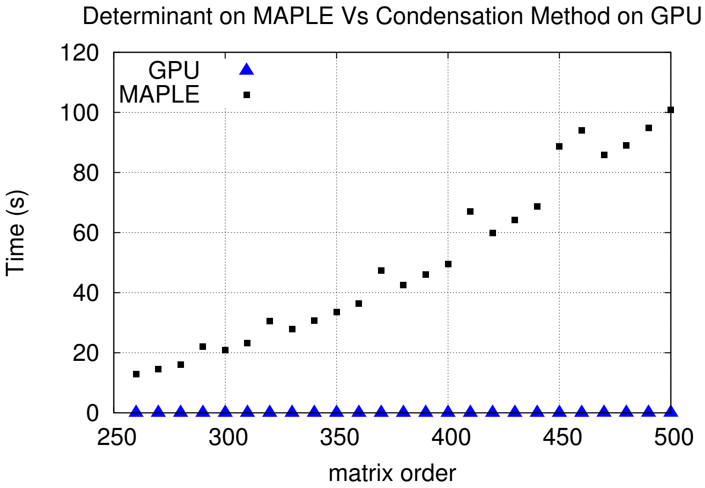
Reference:

NTL: A library for doing number theory, http://www.shoup.net/ntl.

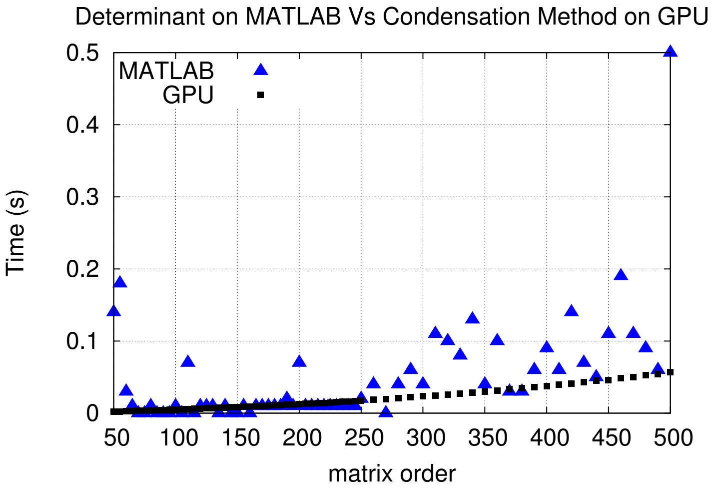
Floating point number Case 1



Floating point number Case 1 (cont.)



Floating point number Case 2



Reference:

Matlab: http://www.mathworks.com.

Hilbert Matrices

- In order to investigate the numerical stability of our GPU implementation of the condensation method, we use the infamous Hilbert matrix $H_{ij} = \frac{1}{i+j-1}$, which is a canonical example of ill-conditioned (and invertible) matrix.
- For example, for $n = 5$, we have

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{bmatrix}$$

Hilbert Matrices (cont.)

Matrix order	MAPLE software floats	MATLAB double floats	Condensation on GPU double floats plus lists
5	0.3239712e-11	3.749295e-12	3.74967e-12
6	-0.1037653175e-16	5.367300e-18	5.36556e-18
7	-0.2940657217e-22	4.835803e-25	4.44292e-25
8	-0.2156380381e-28	2.737050e-33	-3.92813e-33
9	-0.1692148341e-35	9.720265e-43	-2.79235e-41
10	0.4704819751e-42	2.164405e-53	-4.44342e-50
15	0.1386122551e-74	-2.190300e-120	-9.47742e-103
20	0.4711757502e-106	-1.100433e-195	3.81829e-164
25	-0.4092672466-139	5.482309e-274	-3.82134e-239
30	-0.2087134536-174	0	-2.50914e-319
35	0.6863051439e-205	-	3.50293e-398
40	0.3354475665e-237	-	-7.42227e-479
70	-0.1605231989e-443	-	-1.42973e-961
100	-0.1344119185e-667	-	1.96009e-1467
200	-0.1635472167e-1423	-	9.43651e-3169

Table: Determinant of Hilbert Matrix by MAPLE, MATLAB, and condensation method on both CPU and GPU.

Hilbert Matrices (cont.)

Matrix order	MAPLE	MATLAB	Condensation Method on GPU
5	0.004	0	0.000530
6	0.008	0	0.000570
7	0.012	0	0.000595
8	0.008	0	0.000631
9	0.012	0	0.000741
10	0.012	0	0.000447
15	0.016	0	0.000964
20	0.016	0	0.001078
25	0.020	0	0.001271
30	0.024	-	0.001460
35	0.044	-	0.001671
40	0.036	-	0.001896
70	0.188	-	0.003083
100	0.588	-	0.005145
200	5.988	-	0.012488

Table: Time(s) Required to compute determinant of Hilbert Matrix by MAPLE, MATLAB, and condensation method on both CPU and GPU.

Plan

Conclusion

- The condensation method implemented on GPU is a promising candidate to compute determinant of matrices with both modular integer and floating point number coefficients.
- We believe that it can be used to improve the efficiency, in terms of running time and numerical stability, of existing mathematical software packages.

Acknowledgements. We are grateful to Dr. Wei Pan and Dr. Jürgen Gerhard for helpful discussion.

Thank you