Calling the GPU from GNU Octave

Daniel Kraft

WS 2009/2010

1 Introduction

In image processing (and probably in a lot of other applications as well) it is quite commonly necessary to solve sparse linear systems defined in terms of a "stencil" on a 2D grid (that arise from a finite difference discretization of a differential operator like the Laplacian or Biharmonic).

I implemented solvers for a "toy problem" similar to those mentioned above based on iterative schemes (Jacobi iteration and Conjugate Gradient method) on a GPU (using the CUDA programming interface).

One main goal was to allow these solvers to be called from within GNU Octave code, because this RAD (rapid application development) environment is both nice to use for the higher level routines of an image processing application and there's already a lot of existing code that might benefit if some bottle-neck parts could be migrated to compiled code running on a GPU without introducing the need to port the whole system over to C. So this will hopefully also give some general results about the possibility to integrate GPU code with GNU Octave.

2 Problem Statement

For the example problem, we assume that $u : \Omega \subset \mathbb{R}^2 \to \mathbb{R}$ is an image and that we measured $\tilde{u} = Ku + n$, which is blurred (via the operator K) and has noise n added. We want to reconstruct u from $\tilde{u}$ by minimizing this functional:

$$J(u) = \int_\Omega (Ku - \tilde{u})^2 dx + \mu \int_\Omega |\nabla u|^2 \, dx$$

where $\mu > 0$ is a regularization parameter. For my tests I chose $\mu = 10^{-3}$.

It can be found that the necessary optimality condition (i.e., the equation solved by our restored image) is given by this PDE:

$$\begin{cases} -\mu\Delta u + K^2 u &= K\tilde{u} & \Omega \\ \frac{\partial u}{\partial n} &= 0 & \partial\Omega \end{cases}$$

Note that this method of "least squares" is not the very best for image denoising and in fact probably not appropriate for production use because it smoothes the whole image a lot (which is fatal to edges of course, that are prohibited by the $|\nabla u|^2$ term).

In addition, the assumption of a blurred image by operator K is something that's also more of an exercise element than something helping to make this a general-purpose image denoising application — but that's the "toy problem" I worked on for my project anyways. (In fact it's really based on a homework problem from Numerik I by Professor Keeling, which may help to clear some mysteries about choice of some particular details.)

3 Discretization

For a numerical solution (and real images), we consider Ω to be an $N \times N$ grid of pixels, i.e., $\Omega = \{x_1, \ldots, x_N\}^2$ with $x_i = \frac{i}{N}$. I will consider the discretized image u to be a kind of $N \times N$ matrix and denote single pixels by u_{ij}.

In order to take the homogenous Neumann boundary conditions into account, we can add a "margin" of ghost pixels around the real image, such that for instance $u_{0i} = u_{1i}$ and similarly for the other edges.

Then a possible definition of the blurring operator K (and the one I'll use) is to build the weighted mean of the pixel operated on and its four neighbours, like

$$(K_h u)_{ij} = \frac{4u_{ij} + u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}}{8}$$

which can be represented by a "5-point-stencil". Because of the ghost pixels, on the boundary one or two neighbours have by definition the same intensity as the center pixel, and then we get a factor of 5 or 6 instead of the 4.

Note however that K^2 has a much larger stencil — because of this and as K is not very important anyways, I will "approximate" K^2 by K in the optimality system. It would probably be better to set $K = I$ instead for a real world application, but doesn't matter for my purpose of experimenting with GPU solvers; and at least we get some reasonable results this way, too.

Next, we need to discretize the Laplacian. With $h = \frac{1}{N}$, the finite difference approximation I used is given by this:

$$(\Delta_h u)_{ij} = \frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij}}{h^2}$$

where we again have to take the ghost cells into account for pixels on the boundary, but now this has the effect of reducing the factor of 4 to 3 or 2.

This is of course also a "5-point-stencil" and thus the final operator $A = -\mu\Delta_h + K_h$ is again of this form. To find the solution image, we need to solve the linear system $Au = b$ with $b = K_h\tilde{u}$.

So my task was to develop a solver for a sparse linear system defined via a (5-point) stencil. In particular, the exact system has these stencils for inner, edge and corner cells, respectively (possibly rotated to fit the situation):

0	$\frac{1}{8} - \frac{\mu}{h^2}$	0
$\frac{1}{8} - \frac{\mu}{h^2}$	$\frac{4}{8} + \frac{4\mu}{h^2}$	$\frac{1}{8} - \frac{\mu}{h^2}$
0	$\frac{1}{8} - \frac{\mu}{h^2}$	0

$\times$	$\frac{1}{8} - \frac{\mu}{h^2}$	0
$\times$	$\frac{5}{8} + \frac{3\mu}{h^2}$	$\frac{1}{8} - \frac{\mu}{h^2}$
$\times$	$\frac{1}{8} - \frac{\mu}{h^2}$	0

×	×	×
×	$\frac{6}{8} + \frac{2\mu}{h^2}$	$\frac{1}{8} - \frac{\mu}{h^2}$
×	$\frac{1}{8} - \frac{\mu}{h^2}$	0

4 Implementation of Stencil

For solving such a stencil-defined system with an iterative scheme it is important to "know" the stencil (and have the ability to apply its action to some vector) of course. I tried to make the implementation both efficient and flexible, meaning that

a) the values of the stencil (and ideally also its structure) is kept in one place and in a way that makes it easy to change and adapt to other problems with a different stencil, and

b) still allow efficient (i.e., inlined/compiled instead of "interpreted") application and usage of the stencil.

My implementation defines the stencil in a seperate source-file (`stencil_def.inc`) consisting of C preprocessor macro invocations defining the stencil with blocks like this:

```
STENCIL_START(0, row >= 1 && row + 1 < N && col >= 1 && col + 1 < N)
  STENCIL_FIELD(-1, 0, NEIGHBOUR_VALUE)
  STENCIL_FIELD(1, 0, NEIGHBOUR_VALUE)
  STENCIL_FIELD(0, -1, NEIGHBOUR_VALUE)
  STENCIL_FIELD(0, 1, NEIGHBOUR_VALUE)
  STENCIL_CENTER(CENTER_VALUE(4))
STENCIL_END
```

which defines the "inner cell" stencil — that is to be applied if the condition on `row`/`col` of the current cell given in the `STENCIL_START` arguments holds. `NEIGHBOUR_VALUE` evaluates to $\frac{1}{8} - \frac{\mu}{h^2}$ and `CENTER_VALUE(4)` to $\frac{4}{8} + \frac{4\mu}{h^2}$ of course. There's a distinction between the coefficient of the center and of outer fields, because for a Jacobi iteration we need to differentiate between the "diagonal" (i.e., center coefficients) and "off diagonal" elements (i.e., other cells in stencils).

The edge and corner stencils are defined just the same way of course, with corresponding conditions and modified coefficients.

This is (in my opinion) a somewhat compact format to define the stencils; it is obviously not very dynamically modifiable (except if the coefficients refer to variables that are changed, which is a possibility), but that's not needed here anyways. But it can also be turned into "inlined" code where it is used with appropriate definitions for the macros invoked. This (pseudo-)code shows how to do this to implement a step in the Jacobi iteration:

```
procedure jacobi_step (in x[N][N], in b[N][N], out xNew[N][N])
  for row, col = 1, ..., N
    result = b[row][col]

#define STENCIL_START(num, cond) \
```

```
  if (cond)
#define STENCIL_END \
  end if
#define STENCIL_FIELD(dRow, dCol, coef) \
  result -= (coef) * x[(dRow) + row][(dCol) + col]
#define STENCIL_CENTER(coef) \
  result /= (coef)

#include "stencil_def.inc"

    xNew[row][col] = result
  end for
end procedure
```

Similarly it is possible to implement a direct application of the system matrix (defined by the stencils) to some vector as will be needed for the conjugate gradient method.

5 GPU Implementation of the Solvers

As the basic structure of my problem is two-dimensional (processing always takes place on a "vector" representing the pixels of an image), I used also a two-dimensional structure of CUDA threads per block and blocks in the grid.

The basic strategy is then to process a square part of the image in parallel by a block of threads co-operating. After this is done, depending on the problem size and the size / numbers of blocks, the block may go on to do another part of the image. The basic code structure is like this for all kernels I implemented working on the image:

```
const myRow = threadIdx.y
const myCol = threadIdx.x
const firstRow = BLOCK_SIDE * blockIdx.y + myRow
const firstCol = BLOCK_SIDE * blockIdx.x + myCol
const increment = GRID_SIDE * BLOCK_SIDE
const iterations = ceil (N / inc)

for i, j = 0, ..., N - 1
  const row = firstRow + i * inc
  const col = firstCol + j * inc

  ... process image[row][col] ...
end for
```

where `BLOCK_SIDE` and `GRID_SIDE` are the "side-length" of the blocks (in threads) and grid (in blocks) and `N` is the side-length of the image to process.

This strategy (hopefully) allows memory-accesses to be coalesced, because in each block there's a full "column" of threads that access pixels contiguously in memory. Note that the image is stored in column-major order (as is usual for, e.g., Fortran) in memory because this is also what GNU Octave does and I want to interface to that directly.

5.1 Jacobi-Iteration

The implementation of Jacobi iteration for solving the linear system is based on a kernel routine that performs a single iteration; this kernel is then called repeatedly from the CPU, but without transferring data between CPU and GPU, i.e., the image and updated image are always kept in GPU memory only.

For a Jacobi iteration based on the 5-point stencil, each pixel value is needed four times (in calculating the Jacobi update to the four neighbouring pixels) and thus I fetch the old values corresponding to the current thread block in shared memory for faster access to them (of course, more precisely I have to fetch the block plus a margin of extra-pixels around it — i.e., an area of `BLOCK_SIDE + 2` × `BLOCK_SIDE + 2` pixels).

5.2 Conjugate Gradient

I implemented a plain CG without preconditioning for simplicity in this project (but one could have used the system matrix' diagonal as a simple preconditioner, for instance).

At the heart of CG, we have to apply the system matrix to a vector — this is done just in the same structure as the Jacobi iteration described above, because those two operations are indeed very similar (from an implementation point of view).

The other key operations needed are performing an inner product of two vectors, for which I used the CUDA example code, and a simple SAXPY operation ($z = x + \alpha y$) that is implemented very naively with a simple kernel.

With those three kernels available, the CG algorithm itself is run on the CPU, calling the GPU as necessary to perform those operations. Again, while the CPU does all the main "controlling", all vectors are only kept in GPU memory all the time as it is not necessary to transfer them at each step.

6 Interfacing from GNU Octave

GNU Octave allows one to implement functions in C++ via so-called ".oct files" — the concept is basically to write the function using the object-oriented Octave library for accessing the arguments and building the return values as appropriate objects to hand back to the Octave runtime, and this code is then compiled via a special wrapper `mkoctfile` around GCC into a shared library that is loaded and run dynamically by Octave when the function is needed.

On the other hand, in order to use CUDA, the solver code for the GPU has to be compiled by `nvcc`. My strategy was then to build a shared library with the CUDA code by compiling it like this:

```
nvcc -arch sm_13 -shared -Xcompiler '-fPIC' [SOURCES] -o libgpusolver.so
```

where `-arch sm_13` just allows the code to use double precision floating point numbers which is vital because that's also what GNU Octave uses, `-shared` instructs `nvcc` to build a shared library instead of a stand-alone program, and `-Xcompiler '-fPIC'` builds so called "position independent code" that is needed for dynamic linking (at least on 64-bit systems).

Next, I implemented a wrapper function with the Octave interface that accepts and checks the arguments and calls into the solving routines in `libgpusolver.so`. That is, it basically just acts as interface between Octave and the real solver code, handing the data between those two. This can then be compiled by a command like

```
mkoctfile -L. -lgpusolver gpusolver.cpp
```

which builds the file `gpusolver.oct` and allows the function `gpusolver` to be called from within GNU Octave. Note that I had to instruct Octave where to find the shared library (for the GPU code) via setting `LD_LIBRARY_PATH` accordingly, i.e., executing it as

```
LD_LIBRARY_PATH=. octave
```

instead of just `octave`.

7 Results

I tested the code on the Core i7 "compute pc" which has an Intel Core i7 920 Quadcore processor as CPU, 6 GB main memory, a Zotac GeForce GTX 280 GPU, runs 64-bit Ubuntu 9.10 as operating system and GNU Octave 3.0.5.

Because (depeding on the application of course) sometimes absolute precision of solves is not crucial, I chose the tolerance for CG and number of Jacobi iterations always such that the relative error of the iterative solution $\tilde{u}$ to the exact solution u_0 is about $\frac{\|\tilde{u}-u_0\|}{\|u_0\|} \approx 5 \cdot 10^{-4} < 10^{-3}$.

Below are some timing results comparing times for different problem sizes `N` (image is $N \times N$ so we are solving for N^2 unknowns) of solving using Octave's `\` operator on a sparse system matrix, the two GPU implementations and the same solvers implemented on the CPU instead as comparison. The reported time is always the median of three runs, in order to reduce statistical errors. The numbers of threads and blocks to use for the GPU execution is tuned to give empirically optimal performance in the case `N` = 1024 and may thus be suboptimal for smaller problems.

Note that for the "pure Octave" solver building the system matrix alone also takes some time, which is not included in the timings. All overhead of interfacing to Octave as well as data transfer to / from the GPU is however included in the GPU timings (because that's really what a user would experience of course) — this is the reason why especially for small problems the GPU solver is quite slow compared to a direct Octave solve.

N^2	**Octave**	**GPU Jacobi**	**GPU CG**	**CPU GS**	**CPU CG**
16^2	0.8ms	2.4ms	3.1ms	2.2ms	2.2ms
64^2	8.5ms	5.7ms	6.5ms	6.7ms	2.8ms
256^2	279.8ms	381.2ms	90.2ms	1282.8ms	88.4ms
1024^2	8465.2ms	93920.9ms	5091.5ms	346598.0ms	5618.3ms

Here are timings for the GPU CG solver on other problem sizes, that compare the "full" time measured by Octave for the call (including all overheads) to the "raw" time taken by the CG algorithm on the GPU (excluding data transfers

but including of course overheads of kernel calls). Their difference is (mainly) due to data transfer overhead to the GPU but includes also some delay (which is independent of the problem size) because of the time Octave needs for the C++ interface call:

N^2	**Full**	**Raw**	**Overhead**
32^2	5.17ms	2.70ms	2.47ms
128^2	14.38ms	12.31ms	2.07ms
512^2	644.79ms	640.69ms	4.10ms

Thus the difference due to data transfer is insignificant for large problems but makes usage of the GPU for very small problems inefficient (as one could have expected).

8 Conclusion

My results show that it is fairly easily possible to call GPU code from within GNU Octave, but it is only feasible to do so if the problem size is large enough such that the overhead for data transfer to and from the GPU is insignificant. Obviously it is not possible to avoid it (by keeping stuff in GPU memory) across different Octave calls, at least not very easily (an option could be to introduce a kind of "handler" object that stores the pointer to GPU memory or the like).

Of course, the experimentally used Jacobi iteration is a very inferior iteration method, and even Gauss-Seidel with a red-black scheme would probably give much better results. However, the timings of Jacobi on GPU and Gauss-Seidel on CPU (as comparison, with same number of iterations) show that the shared-memory implementation of a Jacobi step based on the stencil is nearly 4 times faster on the GPU, so there's a sensible advantage over CPU here.

For CG, both GPU and CPU are somewhat on par (in my implementation at least) which probably results from the more complicated operations needed to perform and because not only the stencil application makes up most of the time anymore. In addition, the GPU timings of course include the overhead of data transfer, while the CPU can work directly with the memory supplied by GNU Octave without any need to copy; this difference is only sensible for very small problems, though.

But it seems to me that using the GPU similarly to what I tried out may well be a worthwhile option to get rid of bottle-necks in existing Matlab or Octave codes — even when it comes to basic linear algebra stuff; both CPU and GPU solvers based on CG are significantly faster than a direct solve by Octave for large problems (if only an inexact solution is necessary of course), despite the fact, that \ is probably something very optimized (for the general purpose, though).