CUDA and a Red-Black Successive Overrelaxtion Scheme for QUICurb

Andrew Larson*
University of Minnesota Duluth
Last Modified: November 21, 2008

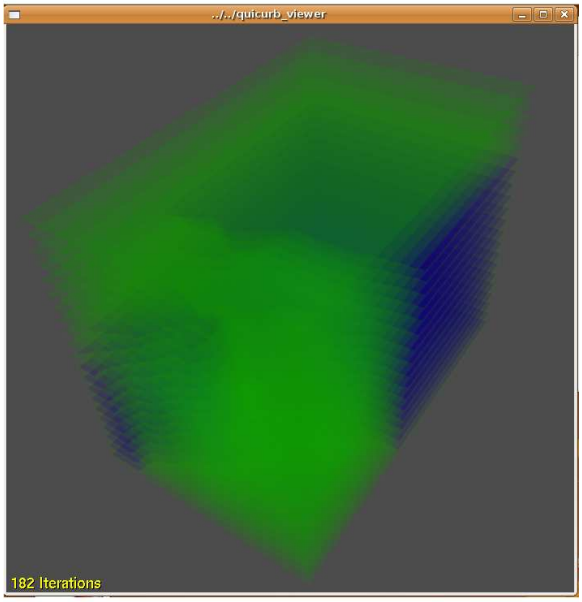


Figure 1: *Viewing slices of an iteration.*

Abstract

The main focus of this paper is discussion concerning the development of a faster computational module in CUDA for computations of the QUICurb simulation.

1 Intro

The Compute Unified Device Architecture, also known as CUDA, is a compiler and set of tools that has made hardware found on NVidia graphics cards available as computation machines. Although GPU core clock speeds do not compare to CPU clock speeds, their parallel design have the potential to provide a faster mechanism for computationly intesive tasks.

1.1 What was done

QUICurb, written in fortran, is executed sequential. CUDA was used to decrease the execution time of the successive overrelaxation scheme used to determine the lagrange multipliers in the finite difference model via parallel execution on the graphics processor. Only one subroutine of the QUICurb source (fortran) was ported to CUDA. This was done as a preliminary step to determine the speedup that could be obtained with CUDA.

2 Background

Mild comprehension of three principle components provide the material and methods used to develop a CUDA solver. These are broad topics that must be handled and do not specifically cover CUDA implementation.

2.1 Successive Overrelaxation

Succesive overrelation (SOR) is a numerical iteration technique used for solving linear systems of equations. It is also known as the accelerated Gauss-Siedel method. The following equation is the formulation of the problem that is used for finding the lagrange multipliers.

$$\begin{aligned}\lambda_{i,j,k}^{t+1} = \omega\frac{-\Delta x^2 R_{i,j,k} + e\lambda_{i+1,j,k}^{t} + f\lambda_{i-1,j,k}^{t+1}}{2\left(o + Ap + Bq\right)} \\ + \omega\frac{A\left(g\lambda_{i,j+1,k}^{t} + h\lambda_{i,j-1,k}^{t+1}\right)}{2\left(o + Ap + Bq\right)} \\ + \omega\frac{B\left(m\lambda_{i,j,k+1}^{t} + \lambda_{i,j,k-1}^{t+1}\right)}{2\left(o + Ap + Bq\right)} \\ + \left(1 - \omega\right)\lambda_{i,j,k}^{t} \qquad (1)\end{aligned}$$

A value of 1.78 is used for ω, where *e*, *f*, *g*, *h*, *m*, *n*, *o*, *p* and *q* are matrices that handle every cell's boundries in the finite difference model. R is the divergence of the observed flow field. $A = \frac{\Delta x^2}{\Delta y^2}$. $B = \frac{\Delta x^2}{\Delta z^2}$. Δx, Δy and Δz represent the cell's size. λ is the matrix of lagrange multipliers, what's trying to be found.

For further information on the development of this particular equation see the QWIC URB Development Notes [Pardyjak 2001]. For

*e-mail: lars2865@d.umn.edu

more on numerical methods see [Atkinson 1989].

2.2 Red-Black

Due to the parallel nature of the solution in CUDA, another technique, red-black, must be applied. The red-black method is common in parallel application of iteration techniques. Because no guarantee during parallel execution can be made to the order in which elements will be evaluated for a given iteration, convergence is not guaranteed nor likely. To compensate, the red-black technique makes two passes for each iteration. (There are other extensions that involve more passes and more decomposition.) If one were to number each element in order, then the first past might calculate all of the new values for the odd elements. Once this is done, the "black", or second pass, could then calculate all the even elements, using the newly updated "red" values. In the next iteration the cycle repeats, where the red elements use the newly updated black values and vice-versa.

The red-black technique provides for a way of extending the original SOR method so the computation can be done in parallel.

2.3 QUICurb

From Pardyjak's [Pardyjak 2001] notes:

> The QWIC URB model is based on the fast response urban wind modeling done in Rockle's Thesis. [Rockle] In this model, an initial uniform wind field is prescribed (u_0, v_0, w_0) based on an incident flow, u_{in} and the various flow effects associated with building geometries. The final velocity field (u, v, w) is obtained by forcing the intial velocity field to be mass consistent. The resulting complex 3D velocity field resembles a time averaged experimental result.

See Pardyjak's notes for further readings. [Pardyjak 2001]

3 How it was done

CUDA uses kernels to perform operations in parallel. The kernels have a grid dimension and block dimension as parameters that serve to break the problem into pieces. In our case the domain is first broken into pieces for each block and further decomposed by distributing the work of a block to individual threads. The grid contains blocks, which contain threads, which are what work on the data in parallel.

Some of the basic kernels that were used and will not be given much discussion include:

Those associated with -

1. zero-ing linear memory
2. finding the max of a matrix
3. finding the min of a matrix
4. mapping matrix entries to an interval

The kernels that will be discussed include:

Those associated with -

1. apiori calculations
2. iteration core
3. total error

3.1 Apriori Denominators

As (1) is written, if one were to implement it directly, then many identical multiplications and additions would be executed each iteration, unnecessarily. To speed calculations the coefficients omega, *A* and *B*, as well as the denominator can be calculated and incorperated into the appropriate matrix *e*, *f*, *g*, *h*, *m* or *n*. This reduces the number of iterations done each iteration, reducing the time to convergence.

The CUDA kernels associated with this process are not complex nor are they what warrants discussion. What is significant is the size of the matrices e, f, g, h, m, n, o, p and q. Currently, each matrice is being stored as a set of floats. Altogether to solve for a discretized domain of 64 x 64 x 20 cells, 4,472,832 bytes are needed, where only 344,064 bytes are needed to store the solution. While 4Mb is not a large amount of memory, 64 x 64 x 20 is not a large domain. When the domain is significantly increased, say to 512 x 512 x 21, the needed amount of memory does the same. With preliminary investigation, assuming that a device, or graphics card, has 512Mb of memory and the domain is increasing in resolution by a factor of 4, a domain of 512 x 512 x 21 will fit into 512Mb of space, but a domain of 1024 x 1024 x 21 will not.

However, if one were to store the boundry values in a more compressed manner, then the amount of needed memory can be reduced significantly. Matrices *e*, *f*, *g*, *h*, *m*, *n*, *o*, *p* and *q* are used to hold the boundry data for all the given cells in the domain. Each element of *e*, *f*, *g*, *h*, *m* and *n* can take on one of 0, .5 or 1. Elements in matrices *o*, *p* and *q* each can be .5 or 1. For any given cell 5832 boundry possibilities exist. Furthermore, all of the possibilities can more than be represented by 2 bytes. Therefore, rather than 9 matrices where each element uses 4 bytes, the boundry conditions can be stored in 1 matrice where each element uses 2 bytes. This storage strategy can significantly reduce the needed memory, allowing for larger domains to fit into smaller locations. The downside is that for each iteration the boundry data must be decoded and redundant calculations done each iteration. Although this will slow the time to convergence, when considering very large domains, ones that would require multiple transfers to and from the host and device, this compression stradegy will prevail. Preliminary calculations suggest that with boundry compression a domain of size 2048 x 2048 x 21 fits into device memory of 512Mb.

Given a constriant on device memory of 512Mb, fitting a significantly larger domain onto the device can be advantageous timewise, since it reduces the number of host-to-device memory transfers. These transfers in a domain of 256 x 256 x 21 accounted for 19.5% of the time to convergence. While a very rough approximation to memory transfer contribution to time, a percentage of this size presents the oppurtunity to make up for time lost during redundant calculations.

Consider a domain of size 2048 x 2048 x 21. Now consider the case when the boundry conditions are compressed and the entire domain fits on the device as opposed to when the boundry conditions are not compressed and only a domian of 512 x 512 x 21 can fit onto the device. For the single transfer needed for a compressed boundry, 16 transfers are needed for the uncompressed version. Because memory transfers will probably contribute a considerable amount of time to convergence, transferring 16 times more per domain or sub-domain will lead to higher times to convergence in the long run. How large does the domain need to be for the compressed boundries to overtake the uncompressed? Approxiamately 3.4 domain transfers. In powers of 2, a domain of 4096 x 4096 x 21 and those that are larger would result in a faster time to convergence for the compressed boundry storage than the uncompressed boundry storage.

The following equations were used to determine the approximate number of domain transfers until the compressed boundry conditions overtake the uncompressed.

$$t_f = 16 * 3600x + ic \tag{2}$$

$$t_c = 1 * 400x + ic + i648 \tag{3}$$

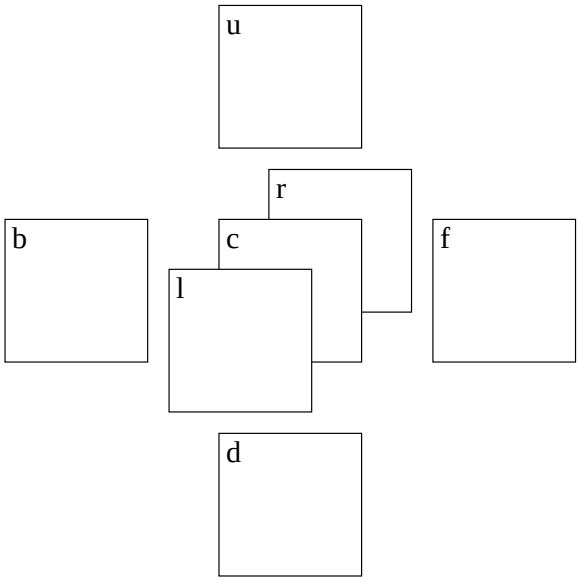
i represents the number of iterations to convergence. *c* represents the base number of clock cycles for each iteration. 648 is estimated number of addition clock cycles needed each iteration for redundant calculations. *x* is the number of domain transfers. 1 * 400 = 400 is the approximate number of clock cycles needed to transfer the compressed data. 16 * 3600 = 57200 is the approximate number of clock cycles needed to transfer the uncompressed data.

This is all speculative. It will be interesting to see if the prediction matches actual results.

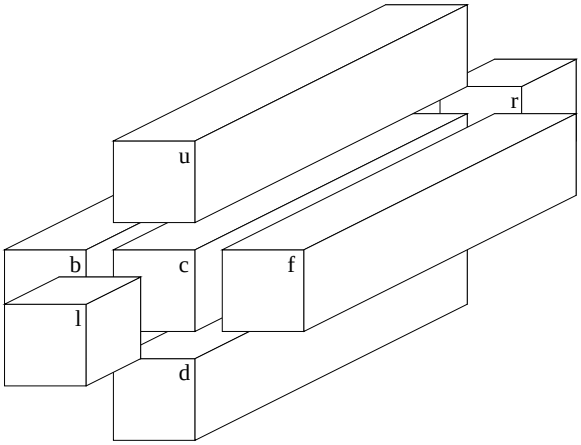
3.2 Iteration Kernel

This is the most complicated kernel and contributes the most to the time to convergence. It is intended to function as one iteration step each time called. This kernel functions similiar to the others, but has a more complicated execution to handle the boundries and access of elements outside of its stick or block space. Data is handled one slice at a time. Slices are anticipated to have 64 cells or more in the x dimension. It is also important that the cells in the x direction are a multiple of 64. (Currently the x and y dimensions are constrained to powers of 2 by the Gather (sum) operation.) Warp sizes are important to consider when using CUDA kernels. This translates into at least a constraint on one dimension that can be built into the kernels to ensure coalesced memory access. If this consideration is not observed, then speed will significantly suffer.

As mentioned, data in the domain is divided by slice. The data division is furthered via operation on blocks within a slice. For instance, if a slice had dimensions 128 x 128 and each block had 64 threads, then the slice would be broken into 256 blocks. The first block, 0, working on the first 64 elements in linear memory, the second block, 1, the next 64 elements, and so on. For any given block the threads that work inside that block have a few tasks. The first task is to load the corresponding data that will be used to calculate the new values for the portion of the domain that these threads are responsible.



If the block has 64 operating threads, then 322 floating point values will be loaded to the block's shared memory for use by these threads, which will only calculate 32 new values. I refer to them as the center stick, the up stick, the down stick, the forward stick, the back stick, the left value and the right value. Each stick in this case would be comprised of 64 floating point values. Stick names were chosen to correspond with the direction from the center stick the grouped values are in the domain. The center stick's values will be updated. All the others are for calculating the new values. Thinking about the domain sliced along the z dimension, the up stick is directly above the center stick in the next highest slice, the down stick directly below. The back stick contains the values with the same offset in the previous row, the front stick the next row. The left and right values those directly to the left of the center stick and directly to the right.



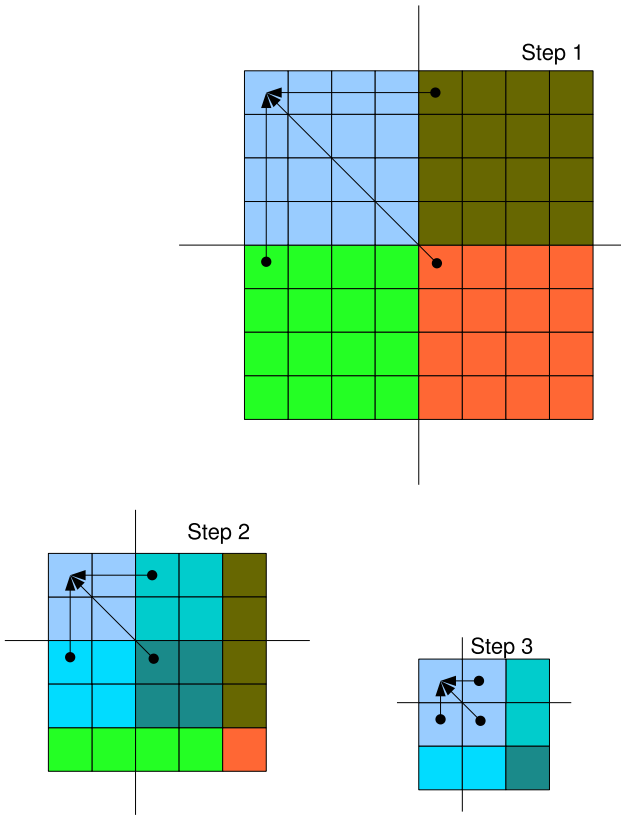
Only 32 new values will be calculated. This is due to the red-black technique used. On each pass 32 of the threads will determine that they are not of the current pass and should not calculate updated values. Here is an area that may provide more speed by only using half as many threads, only those needed. It's posssible that the number of concurrent operations would increase. Currently, in a block with 64 threads, 128 floating point values will not be accessed and used for calculations. These values are in the sticks surrounding the center stick. Those values account for roughly four tenths of the data read to shared memory. Again, another area that may squeeze just a little more speed out.

This approach is more complex when any part of the center stick contains boundry values. The top and bottom slices are excluded from the calculations through the wrapper function. Threads will not work on the bottom or top slice. For the first and last row of a particular slice, if the threads determine that they are in that row, then they simply return without loading or modifying anything. If a thread is in the first column or last column of any of the remaining rows, then that thread will return after loading its respective value. In this way, the boundry conditions are nearly entirely built into the iteration kernel. The boundry condition that assigns values of slice 1 to slice 0 is handled seperately. The integrated boundries may be an area to obtain more speed.

3.3 OLD - Gather (sum) Kernel

The summing kernel is straight forward, but still ranks number 2 in amount of time contributed to the time to convergence. This kernel should be the next to get an overhaul since it constrians the domain to powers of 2 in both the x and y dimensions. The kernel adds all the elements in a slice, leaving the sum in location (0,0,k), where k is between 0 and the number of cells in the z dimension.

Elements are summed by splitting a slice into 4 equal parts, adding the 4 corresponding elements from equal part together and storing the result in 1 of the parts. This is done repeatedly until only one element has the sum of the entire slice.



For any set of data there is a threshold where occupancy on the device drops, uncoalesced memory accessed take over and times take a small step up from the last and do not decrease nearly as fast as before the break. This threshold is likely related to warpsize and may not be avoidable.

3.4 NEW - Gather (sum) Kernel

The summing kernel has been overhauled. It still requires the second most amount of processing. However, the domain dimension limits have been removed. This kernel can sum any linear memory space that has less than 67,107,840 elements. This number is related to the number of threads used, the use of 4 section chunks and the maximum thread dimension in the grid, i.e. 256 * 4 * 65535.

For a given section in linear memory, its size and a pointer to a device pointer for output, the kernel will place the total sum in to the memory pointed by the last device pointer passed, the third argument. This is done by first finding the number of thread blocks needed to cover the majority of memory. Anything that is left over is done slowly by one incomplete thread block with the left overs, which should be small compared to the number of elements added in parallel.

In each block 256 threads are used, each of which loads 4 elements into shared memory. Each block of threads is responsible for 1024 elements, where the threads each load 4 elements into 4 sections. A miniature reduce is done in each block. To start, each of the threads sums four elements, one from each section (256 elements) and places it in to an device array of sums. Originally, sections are determined by pointers. These pointers are reassigned after each reduction step. Originally offset by 256 from one another, in the second reduction step the section pointers will be offset by 64, with all of them now pointing into the first section of the 1024 block. Each of the threads sums four elements and stores it in the first 64 locations of the first section. This is continued until the first element of the first section contains the sum of this block. This element is then placed in a temp device array.

Once all of the blocks have done there reductions on their specific section of memory, the results are added either serially, or if there are enough blocks, another reduction is done. Finally, a serial add is done on the remaining sums to get the final sum, placed into the passed device pointer.

A note of interest: no thorough comparison was done, but the provided Nvidia reduction (with addition) appears to be 8x faster than this one, has domain restrictions and is very complicated looking.

The min and max kernels use the same general thread blocking and reduction via section pointer approach.

3.5 Absolute Difference kernel

Although this kernel is very straighforward, the block allocation is worth mentioning. Very much derived from the sum kernel thread blocking, these kernels launch *domain size / threads + 1* blocks. Currently, the thread count is 512 which allows for finding the difference of domains with 512 * 65535 = 33,553,920 elements. Not done, but a simple modification would allow for larger domains with more calls to the kernel, using the proper memory offset. This is a place where streaming could take place.

With the current implementation two kernel launches take place. One that launches enough thread blocks to cover a majority of the elements. And a second that convers the remaining elements that are not enough to make up a thread block.

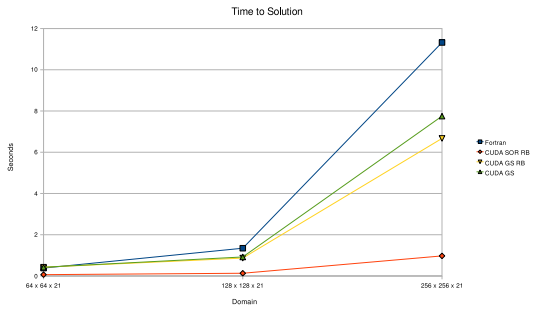
The zero, relative difference, denominators and map to interval kernels all use this same approach.

4 Conclusion

CUDA provides a plausible means for increasing not only the speed of calculating the lagrange multipliers but also the speed of the entire QUICurb module. Another advantage to porting fortran to CUDA is the integrability of CUDA with C++, which allows for a much more module solution to the problem.

4.1 Pay-offs

Some comparisons have been done between code written in fortran that uses the SOR technique with double precision, code written in C++ that uses single precision, code written in CUDA that uses a Gauss-Siedel Red-Black approach with single precision, code written in CUDA that uses a Gauss-Siedel approach with single precision and code written in CUDA that uses a SOR Red-Black technique with single precision. The following graph compares four of the five and their time to convergence across different domain sizes.



Using CUDA with the a Red-Black SOR technique clearly out preforms timings using fortran as domain sizes grow.

These numbers are limited by domain size options that the current CUDA setup can run. A more thorough investigation involving more domain sizes should be done, which requires a more generic solution in CUDA.

The more thorough investigation can be done now. Domain has opened to only nx being restricted to multiples of 64.

4.2 Future Directions

4.2.1 Modifications

Restating areas mentioned before, which can and/or should be changed:

1. Tweak iteration core for better performance. (largest convergence time contributor)
   (a) 2x More threads allocated to a block then do calculations
   (b) Most of the boundries are integrated
   (c) Modulus is expensive, find alternatives
2. Compress the boundry condition matrices.
   (a) Will improve super-large domain times
   (b) Will allow for larger domains without significant modifications
   (c) Reintroduces redundant calculations
3. Iteration Viewer
   (a) Include more information in display

4.2.2 Fortran to CUDA

1. Incorperate the rest of the QUICurb module.
   (a) building_parameterizations.f90
      i. defbuild.f90
         A. pentagon.f90
      ii. upwind.f90
      iii. street_intersect.f90
      iv. plantinit.f90
         A. regress.f90
      v. wake.f90
         A. building_connect.f90
      vi. rectanglewake.f90
      vii. cylinderwake.f90
      viii. bridgewake.f90
      ix. streetcanyon.f90
      x. rooftop.f90
      xi. courtyard.f90
      xii. parking_garage.f90
      xiii. build_garage.f90
      xiv. wallbc.f90
      xv. poisson.f90
      xvi. unbuild_garage.f90
   (b) sensorinit.f90
      i. read_quic_met.f90
      ii. readittmm5_met.f90
      iii. read_hotmac_met.f90
   (c) sort.f90
   (d) init.f90

References

ATKINSON, K. E. 1989. *An Introduction to Numerical Analysis*. John Wiley and Sons.

PARDYJAK, E. R. 2001. QWIC URB - Development Notes.

ROCKLE, R. PhD thesis.