

Obtaining a 35x Speedup in 2D Phase Unwrapping Using Commodity Graphics Processors

Peter A. Karasev[†], Daniel P. Campbell*, and Mark A. Richards[†]

[†]School of Electrical and Computer Engineering, *Georgia Tech Research Institute
Georgia Institute of Technology
Atlanta, GA 30332-0765 USA

Abstract - Graphics processing units (GPUs) are a powerful tool for numerical computation. The GPU architecture and computational model are uniquely designed for high-resolution high-speed grid-based calculations. This capability can be utilized to accelerate certain classes of compute-intensive radar signal processing algorithms. Characteristics of a problem well-suited for computation on a GPU include high levels of data parallelism, low control logic, uniform boundary conditions, and well-defined input and output.

We describe the implementation of two-dimensional multigrad least-squares weighted phase unwrapping on a GPU and demonstrate a large speedup over C and MATLAB implementations. Details of the GPU computation are provided. Background information on the GPU architecture and its applicability to general-purpose computation is discussed.

I. INTRODUCTION

Graphics processing units (GPUs) are application-specific processors that are used to accelerate the standard three-dimensional graphics rendering pipeline. Over the past several years, they have become increasingly programmable and as a result have become useable for some scientific computations.

GPUs can execute floating point operations an order of magnitude faster than standard CPUs. Current generation GPUs have delivered observed performance of up to 250 GFLOPS on synthetic benchmarks, in comparison to a peak theoretical computation rate of approximately 28 GFLOPS for a dual core Pentium 4, 3.46 GHz CPU. Memory bandwidth is also significantly greater in the graphics processor than between a conventional CPU and its main memory, thus avoiding another major bottleneck in large computational problems. GPUs achieve memory bandwidths on the order of 35 GB/s, while CPUs are limited to approximately 6 GB/s [1]. Furthermore, these performance advantages are increasing due to intense competition in the GPU market. Finally, GPUs are economical: a top-tier GPU can be purchased for under \$500, significantly less than a DSP or FPGA chip offering similar performance.

This work was supported by the U.S. Defense Advanced Research Projects Agency under U.S. Air Force contracts FA8750-06-1-0012 and F30602-02-2-0124.

The programmability of the most recent generation of GPUs creates the opportunity to develop very powerful, low cost accelerators for key radar signal processing algorithms. In this paper, we describe an experiment in the application of GPUs to the two-dimensional phase unwrapping problem at the heart of interferometric synthetic aperture radar (IFSAR) processing. While phase unwrapping is relatively simple in one dimension, it becomes quite complex in multiple dimensions. One approach to the problem of recovering the original phase from a measurement of wrapped phase in the presence of noise or other distortions casts it into the mathematical framework of a solution to the discretized Poisson's equation [2]. The resulting least squares solution for the weighted phase unwrapping problem involves the use of an iterative solution technique requiring only scalar add, subtract, multiply, and divide operations at each step, and is well-suited to GPU implementation.

II. USING GPUS FOR SCIENTIFIC COMPUTATION

The primary market application of the GPU is computing pixel intensities and colors for high resolution video games and CAD tools. The most computationally intensive portion of the image rendering pipeline is a step where nearly identical, mutually independent and spatially distributed operations are performed on each pixel. Consequently, GPUs emphasize increasing the degree of data parallelism implemented in the processor. Current GPUs have 32 or more parallel pixel pipelines, with more being added in each new generation.

Figure 1 diagrams the standard GPU processing pipeline, which contains two stages that can be used for general purpose scientific computation. The most easily programmable part of the pipeline, used here for phase unwrapping, is the final section called the *fragment processor*. A *fragment* is a pixel that can potentially be rendered to the screen. A *texture* is an array of data that can be used in a rendering operation. In the fragment processor, numeric calculations are performed on a per-pixel basis. Thus, the fragment processor can be used to perform data parallel operations on a 2D data set.

Recent advances in programmable GPUs have made their computational power much more accessible. For example,

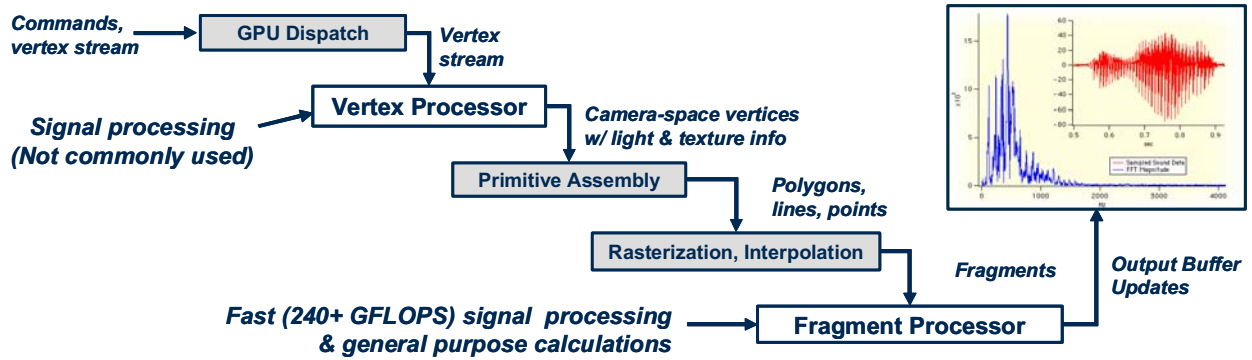


Figure 1. Diagram of the GPU Rendering Pipeline as applied to General Purpose Computation.

prior generations of GPUs and their application programming interface (API) required that all data be 8-bit fixed point values, but GPUs now provide for 32-bit floating-point calculation in off-screen buffers. While much recent work on general purpose computation using GPUs relied on assembly-level shader programs, this work uses the higher-level Cg (“C for Graphics”) language from NVIDIA. The input to a Cg program can consist of a number of large arrays (implemented as textures) or smaller structures such as scalars or three-vectors. Cg is similar to C in terms of syntax, but it also supports additional instructions that have dedicated hardware support, such as vector dot products and cross products or matrix-vector multiplication for dimensions up to four.

Because of the structure of the GPU pipeline, its use for efficient general purpose computation is confined to algorithms that perform a very similar operation on a number of data points in an array (texture). The process of applying the mathematical operations to each data point (pixel in the texture) is called a *rendering pass*. Other output patterns are possible but require additional time or rendering passes, reducing the overall performance improvements gained by using the GPU.

III. WEIGHTED LEAST-SQUARES 2D PHASE UNWRAPPING

The phase of a complex pixel $z_{m,n} = A_{m,n}e^{j\phi_{m,n}}$ (where m,n denotes the sampled image coordinates in the x and y dimensions) in a SAR image is related to the range to the imaged pixel measured in wavelengths, and will typically be a very large number. However, in the signal processor the phase is computed from the real and imaginary parts of z using the arctangent function. Consequently, the measured phase $\psi_{m,n}$ is “wrapped”, that is, bound to the range $[-\pi, \pi]$. The wrapped phase values are related to the actual phase values $\phi_{m,n}$ according to:

$$\psi_{m,n} = \phi_{m,n} + 2\pi k, \quad (1)$$

where k is an unknown integer.

Figure 2 is a simple illustration of phase wrapping. Figure 2(a) is a phase pattern with a total range of 63 radians. Every time the actual phase exceeds π radians, the phase as measured using the arctangent of the real and imaginary parts of the data value “wraps around” to $-\pi$ radians, creating the interferometric “fringes” of Fig. 2(b). This wrapped phase represents the available data in a phase unwrapping problem. The goal of the unwrapping algorithm is to reconstruct the unwrapped phase function of Fig. 2(a).

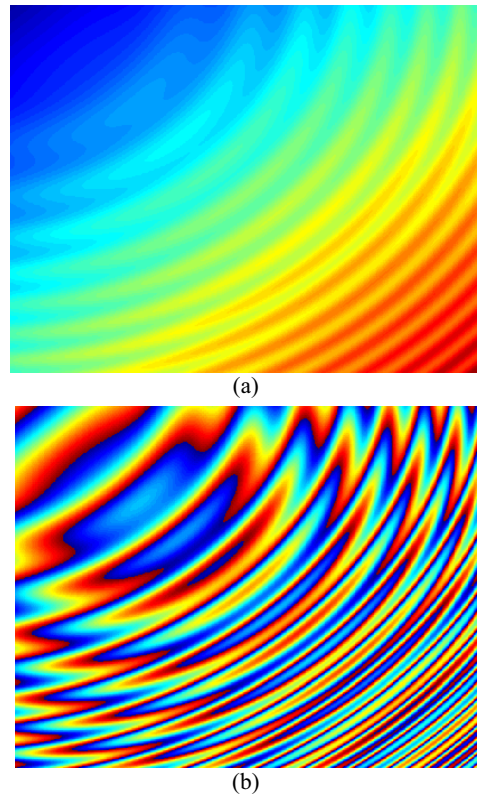


Figure 2. (a) Original phase function, increasing from upper left to lower right over a total range of 63 radians. (b) Wrapped version of (a), showing the expected cycles through the range $[-\pi, \pi]$.

One algorithm for doing so is the weighted least squares phase unwrapping algorithm, described in detail in [3]. In brief, the version of the algorithm considered here proceeds as follows. First, gradients $\Delta_{m,n}^x$ and $\Delta_{m,n}^y$ in the x and y directions are estimated using wrapped first differences. The algorithm then seeks to minimize the error between the gradient of the unwrapped and wrapped phase estimates, subject to pre-calculated weightings based on data quality at each point. Specifically, we seek to minimize

$$\varepsilon^2 \equiv \sum_{m,n} U_{m,n} (\phi_{m+1,n} - \phi_{m,n} - \Delta_{m,n}^x)^2 + \sum_{m,n} V_{m,n} (\phi_{m,n+1} - \phi_{m,n} - \Delta_{m,n}^y)^2 \quad (2)$$

where $U_{m,n}$ and $V_{m,n}$ are weighting values that are typically related to signal-to-noise ratio. This weighted Poisson's equation can be solved iteratively using a classical Gauss-Seidel or Jacobi solver, which iterates the equation

$$\phi_{m,n} = \left(U_{m,n} \phi_{m+1,n} + U_{m-1,n} \phi_{m-1,n} + V_{m,n} \phi_{m,n+1} + \dots + V_{m,n-1} \phi_{m,n-1} + \rho_{m,n} \right) / \left(U_{m,n} + U_{m-1,n} + V_{m,n} + V_{m,n-1} \right) \quad (3)$$

The iteration is considered to have converged when the change in the mean-square difference in unwrapped phase between iterations falls below a pre-set threshold.

IV. MULTIGRID ACCELERATION

A technique called *multigridding* can be used to accelerate the convergence of the basic iteration of (3). Multigridding starts by reducing the original problem to a lower resolution problem on a smaller grid. Once the low resolution solution is obtained, the result is upsampled to a larger grid and a new, higher resolution solution computed. This process is repeated until a solution is obtained at the original grid size and resolution. This technique decreases the fundamental one-grid-square-per-iteration propagation of the solution in (3). The specific algorithm used here proceeds as follows for a problem defined on an $N \times N$ grid:

1. Downsample the data to an $(N/4 \times N/4)$ grid.
2. Iterate (3) until meeting a convergence criterion or running a maximum number of iterations.
3. If the current grid size $M \times M$ is less than $N \times N$, upsample to a $(2M \times 2M)$ grid and go to step 2. Otherwise, terminate the algorithm.

The multigrid method speeds up resolution of low frequency components on the smaller grid sizes, while iteration at the full-sized grids resolves high-frequency components [3]. The benefit of multigridding is limited in IFSAR phase unwrapping in comparison to some other numerical partial differential equation (PDE) applications due to the relatively large amount of high frequency components in typical terrain maps. This means that any solution method

must necessarily spend substantial time on the larger grid sizes. Because of the high data parallelism of the GPU, this characteristic actually further increases the advantage of the GPU over the other implementations for phase unwrapping.

The use of multigridding provides significant speedup in the IFSAR phase unwrapping as compared to the original non-multigrid algorithm. All of the results reported in this paper are for the multigrid algorithm with a minimum grid size of $N/4 \times N/4$. $N/4$ was chosen as the minimum grid size because further reductions are likely to be impractical for the SAR phase unwrapping problem due to loss of resolution.

V. IMPLEMENTATION OF PHASE UNWRAPPING ON A GPU

The rectangular setup of the area for solution of the weighted Poisson's equation and the iterative solution technique are an excellent match for the GPU computational model. A program to implement the weighted least squares phase unwrapping program was written in NVIDIA's Cg (C for Graphics) language. The inputs to the GPU solver are the gradient estimates $\Delta_{m,n}^x$ and $\Delta_{m,n}^y$, the weighting matrices $U_{m,n}$ and $V_{m,n}$, and the problem dimension N . The implementation was verified to be capable of accepting various weights, such as a function of phase-derivative variance or magnitude of signal. For the artificial data of these experiments the weights were set to 1.0 but were still explicitly applied so that they contributed to the computational load. After initialization, in each render pass the program performs one weighted Jacobi iteration. The process continues until convergence, which is checked by periodically reading the data back to the CPU from the graphics processor.

The program was implemented on a dedicated AMD Athlon64 PC using Windows XP with an NVIDIA Geforce 8800GTX video card having 768 MB video RAM and 1 GB of system RAM. Simulated data were created using a variety of two-dimensional phase functions that were then wrapped and used as input. The algorithm was run for data grids ranging in size from 256×256 to 2048×2048 . Define the mean-square difference of the unwrapped phase from one read-back to the CPU to the next as

$$\varepsilon = \sum_{grid} \left(\phi_{m,n}^k - \phi_{m,n}^{k+r} \right)^2 \quad (4)$$

where r is the number of iterations between read-backs. The algorithm is then assumed to have converged when the problem is upsampled to the largest grid size and

$$\varepsilon < 10^{-5} N^2 \quad (5)$$

As can be seen from Table I, the number of iterations required ranged from about $25N^2$ to about $50N^2$ on the grid sizes tested. The gross structure of the unwrapped phase estimate is largely defined at the smaller grid sizes, while the largest grid size refines the high resolution detail. The exact convergence time can vary significantly depending on the

TABLE I

TIME (SECONDS) AND ITERATIONS TO OBTAIN CONVERGENCE IN PHASE UNWRAPPING

	256x256	512x512	1024x1024	1600x1600	2048x2048
GPU	0.156	0.672	5.69	22.22	59.9
C on CPU	0.828	15.125	194.4	815.5	2085.02
MATLAB	17.26	291.5	6886.6	213.7e3	453.9e3
Iterations	2500	10000	30000	50000	80000

TABLE II

POINT UPDATES PER SECOND ON VARYING GRID SIZES

	256x256	512x512	1024x1024	1600x1600	2048x2048
GPU	3.41e8	1.27e9	1.80e9	1.87e9	1.82e9
C on CPU	6.43e7	5.63e7	5.26e7	5.10e7	5.23e7
MATLAB	4.15e6	2.92e6	7.23e5	1.35e5	1.66e5
Iterations	2500	10000	30000	50000	80000

particular input wrapped phase function. For each grid size in these experiments, an identical input and number of iterations was used for each implementation.

For comparison purposes, the same algorithm was implemented in two alternate versions that did not utilize the GPU. The first was coded in conventional ANSI C, and compiled with the Microsoft Visual Studio 7.1 compiler in release mode. The second version was implemented in MATLAB for Windows version R14. These versions were run on the same computer as the GPU implementation. Reasonable care was taken to optimize all three versions, but extensive optimizations that substantially alter the nature of the source code, such as incorporating assembly language modules for key operations in the MATLAB and CPU versions, were not undertaken.

VI. RESULTS

For an end user, the most important metric is the total run time to achieve a solution. Table I compares the time to convergence for the MATLAB, C, and GPU implementations for grid sizes from 256x256 to 2048x2048. Figure 3 is a plot of the data in Table I. Due to excessive run times, the MATLAB execution times in the shaded boxes were extrapolated by measuring the time required to execute the first 10% (for 512x512), 1% (for 1024x1024) or 0.1% (for 1600x1600 and 2048x2048) of GPU and C iterations.

The GPU implementation demonstrates substantially better performance than the other two versions, especially with larger grid sizes. For example, on a 256x256 grid the GPU implementation requires 0.156 seconds to converge, as compared to 0.828 for the C implementation of the same algorithm and a slow 17.26 for the MATLAB version. This

corresponds to a GPU speedup of 5.3x relative to the C version and 110.6x relative to MATLAB. As the grid gets larger, the GPU scales better; on the 2048x2048 grid, the speedup using the GPU relative to the C implementation is a factor of 34.5x. Similar speedups are seen for the other large grid sizes; the GPU consistently provides well over an order-of-magnitude improvement in run time as compared to conventional C.

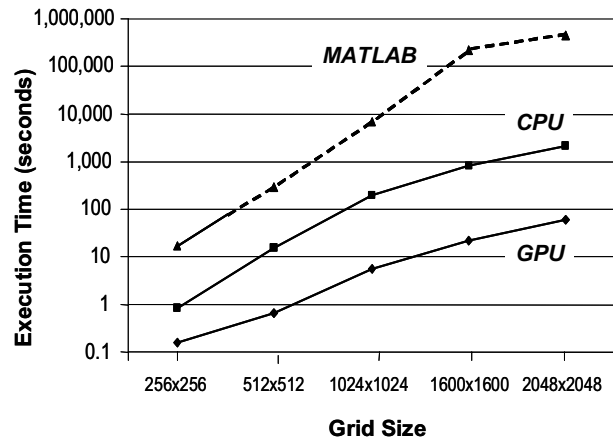


Figure 3. Comparison of phase unwrapping solution time for C, MATLAB, and GPU using identical algorithm. Dotted line indicates extrapolated values.

Table II compares the implementations in terms of the number of “point updates” per second; this data is plotted in Fig. 4. A point update is an operation to perform (3) at one grid point during one iteration; this metric takes into account the varying grid sizes during the multigrid algorithm’s runtime

and becomes nearly constant once the grid is large enough to saturate all caches on a given architecture (this is not the case for a software-based environment like MATLAB which degrades further with grid size). For example, the 256x256 procedure takes 2500 iterations (1000 iterations on the 64x64 grid, 1000 on 128x128, and 500 on 256x256), so the total number of point updates is $(64)(64)(1000) + (128)(128)(1000) + (256)(256)(500) = 53,248,000$.

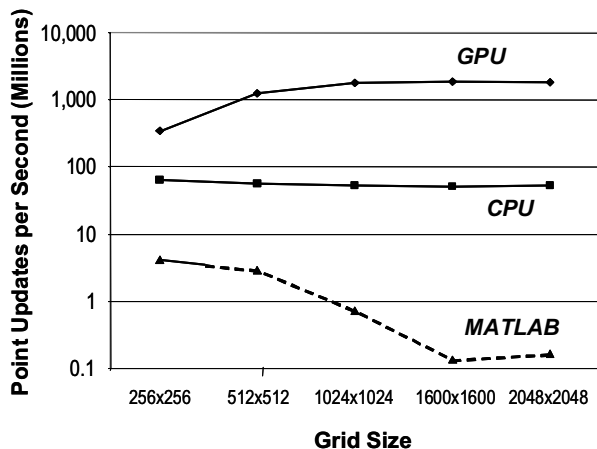


Figure 4. Comparison of point updates per second for C, MATLAB, and GPU on the phase unwrapping algorithm. Dotted line indicates extrapolated values.

This metric gives a more general sense of the relative computational advantage of the GPU as it might be applied to other radar signal processing algorithms. These per-point speedups are the fundamental reason for the advantage of the GPU implementation of the phase unwrapping algorithm.

VII. CONCLUSION

We have demonstrated that speedups in the range of 5x to 35x are possible in 2D multigrid weighted least squares phase unwrapping on reasonable grid sizes by using GPUs as computational accelerators. GPUs are ubiquitous, inexpensive, and yet increasingly powerful processors. At the same time, they require special programming techniques and are most effective only for algorithms that fit their computational model well.

The two-dimensional IFSAR phase unwrapping problem, using the iterative weighted least-squares solution technique, has a number of characteristics that make it well-suited to acceleration using a GPU:

1. It contains only floating-point operations and array indexing (no integer computation).
2. The array of values being updated is used in each iteration and each of its constituent elements has a consistent differential relationship with respect to itself and its neighbors (*i.e.*, the solution procedure is amenable to being described in a form like (3)).
3. The GPU provides specific hardware support for the required boundary conditions, which have zero spatial derivatives along the normal to the grid boundary. Other uniform boundary condition types would also be straightforward to implement.
4. There is no control logic for “on the fly” changes of computation procedure.
5. Operations are performed per-point, *i.e.*, there is no computation performed that requires reading across the entire grid at each iteration before making a compute and write operation.
6. The procedure is highly memory-intensive at larger grid sizes.

Many radar signal processing algorithms may have compute-intensive sections with similar characteristics that can utilize the GPU as a coprocessor and take advantage of the rapid performance increases and low costs of GPU technology. Additionally, many computations outside of signal processing can readily be implemented. One very straightforward application would be a linear sweep to calculate S parameters, with each pixel representing a frequency with a constant Y matrix computation at each point. Finite difference methods are also a subject of research on the GPU [1]. Finally, developments in the GPU industry, such as a move toward a simplified API that bypasses the current need to use OpenGL or DirectX to communicate with the hardware and makes more parallel pipelines accessible, will further increase the applicability of these devices.

REFERENCES

- [1] Taflov, Allen and Susan C. Hagness. *Computational Electrodynamics: the Finite-Difference Time-Domain Method*. Boston: Artech House, 2005.
- [2] Ghiglia, Dennis and Louis A. Romero. “Robust two-dimensional weighted and unweighted phase unwrapping that uses fast transforms and iterative methods”, *J. Opt. Soc. Am. A* 1, 107-117 (1994).
- [3] Ghiglia, Dennis and Mark D. Pritt. *Two-Dimensional Phase Unwrapping: Theory, Algorithms, and Software*. New York: Wiley-Interscience, 1998.