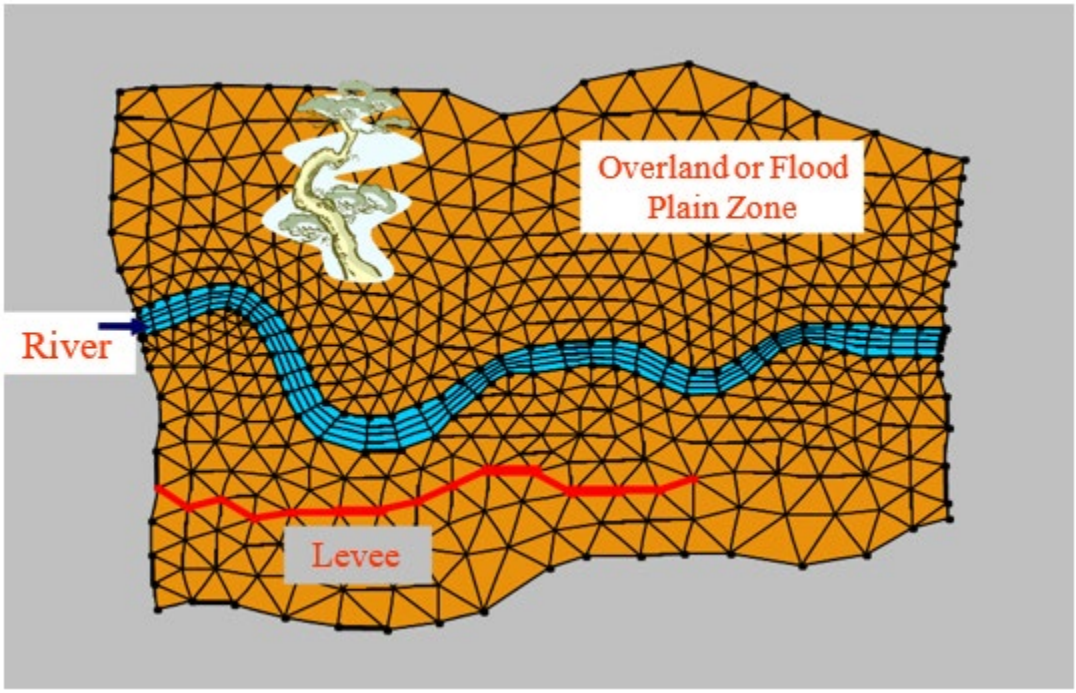
BUREAU OF RECLAMATION

Development of a GPU Accelerated Salinity Module for the SRH-2D Platform

Science and Technology Program
Research and Development Office
Final Report No. ST-2020-1883-01



U.S. Department of the Interior

March 12, 2021

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.			
1. REPORT DATE *(DD-MM-YYYY)* 12-03-2021	2. REPORT TYPE Research		3. DATES COVERED *(From - To)* 2017-10-01 – 2020-09-30
4. TITLE AND SUBTITLE Development of a GPU Accelerated Salinity Module for the SRH-2D Platform			5a. CONTRACT NUMBER 20XR0680A1- RY15412018WP31883 5b. GRANT NUMBER NA 5c. PROGRAM ELEMENT NUMBER 1541 (S&T)
6. AUTHOR(S) Vanessa King, Hydrologist			5d. PROJECT NUMBER Final Report No. ST-2020-1883-01 5e. TASK NUMBER NA 5f. WORK UNIT NUMBER NA
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Planning Division, Decision Analysis Branch California-Great Basin Regional Office Bureau of Reclamation U.S. Department of the Interior 2800 Cottage Way, W-2830 Sacramento, CA 95825			8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Science and Technology Program Research and Development Office Bureau of Reclamation U.S. Department of the Interior Denver Federal Center PO Box 25007, Denver, CO 80225-0007			10. SPONSOR/MONITOR'S ACRONYM(S) Reclamation 11. SPONSOR/MONITOR'S REPORT NUMBER(S) Final Report ST-2020-1883-01
12. DISTRIBUTION/AVAILABILITY STATEMENT Final Report may be downloaded from https://www.usbr.gov/research/projects/index.html			
13. SUPPLEMENTARY NOTES			
14. ABSTRACT Reclamation's mission is designed around delivering usable agricultural water for the West. Therefore, being able to simulate salinity in a 2-D finite-volume model in a realistic timeframe would be of significant value to Reclamation. The goal of this project was to develop a salinity module to simulate salinity using SRH-2D, a two-dimensional (2D) flow hydraulic and mobile-bed sediment transport model for river systems that is frequently used by Reclamation. Additionally, this research sought to improve the efficiency of SRH-2D using graphics processing unit (GPU) acceleration methods. Unfortunately, neither goal was achieved during the timeframe of this project, but the research laid the groundwork for future research in this area.			
15. SUBJECT TERMS Water quality, modeling, salinity			
16. SECURITY CLASSIFICATION OF:		17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES 19a. NAME OF RESPONSIBLE PERSON
a. REPORT U	b. ABSTRACT U	THIS PAGE U	19b. TELEPHONE NUMBER *(Include area code)*

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Mission Statements

The Department of the Interior (DOI) conserves and manages the Nation's natural resources and cultural heritage for the benefit and enjoyment of the American people, provides scientific and other information about natural resources and natural hazards to address societal challenges and create opportunities for the American people, and honors the Nation's trust responsibilities or special commitments to American Indians, Alaska Natives, and affiliated island communities to help them prosper.

The mission of the Bureau of Reclamation is to manage, develop, and protect water and related resources in an environmentally and economically sound manner in the interest of the American public.

Disclaimer

Information in this report may not be used for advertising or promotional purposes. The data and findings should not be construed as an endorsement of any product or firm by the Bureau of Reclamation, Department of Interior, or Federal Government. The products evaluated in the report were evaluated for purposes specific to the Bureau of Reclamation mission. Reclamation gives no warranties or guarantees, expressed or implied, for the products evaluated in this report, including merchantability or fitness for a particular purpose.

Acknowledgements

The Science and Technology Program, Bureau of Reclamation, sponsored this research. Yong Lai provided technical support and expertise.

Development of a GPU Accelerated Salinity Module for the SRH-2D Platform

Final Report No. ST-2020-1883-01

prepared by

California-Great Basin Regional Office
Vanessa King, Hydrologist

Cover Image: Illustration of zonal partition and mesh layout in SRH-2D. From Lai (2008).

Peer Review

Bureau of Reclamation
Research and Development Office
Science and Technology Program

Final Report ST-2020-1883-01

Development of a GPU Accelerated Salinity Module for the SRH-2D Platform

VANESSA KING Digitally signed by VANESSA KING Date: 2021.03.12 08:08:18 -08'00'

Prepared by: Vanessa King
Hydrologist, California-Great Basin Region Planning Division

YONG LAI Digitally signed by YONG LAI Date: 2021.03.17 16:07:01 -06'00'

Peer Review by: Yong Lai
Hydraulic Engineer, Technical Service Center

Acronyms and Abbreviations

1D	One-dimensional
2D	Two-dimensional
3D	Three-dimensional
ADI	Alternating Direction Implicit
AS	Additive Schwarz
BiCGSTAB	Bi-Conjugate Gradient Stabilized
BLAS	Basic Linear Algebra Subroutines
BSR	Block Compressed Sparse Rows
BSRX	Extended Block Compressed Sparse Rows
CFL	Courant-Friedrichs-Lewy
CG	Conjugate Gradient
CR	Cyclic Reduction
COO	Coordinate
CPU	Central Processing Unit
CSC	Compressed Sparse Columns
CSR	Compressed Sparse Rows
CUDA	Compute Unified Device Architecture
DAG	Directed Acyclic Graph
DEM	Digital Elevation Model
DIA	Diagonal
ELM	Eulerian-Lagrangian Method
FFT	Fast Fourier Transform
GMRES	Generalized Minimum Residual
GPU	Graphics Processing Unit
HEC	Hybrid ELL and CSR
HDI	Hacked Diagonal
HYB	Hybrid
IC	Incomplete Cholesky
ILU	Incomplete LU
INVK	Positional Fill Level Triangular Inverse
INVT	Numerical Fill Drop Triangular Inverse
JAD	Jagged Diagonal
LLK	Left-looking AINV
MKL	Multiple Kernel Learning
PCR	Parallel Cyclic Reduction
PDE	Partial Differential Equation
PSBPLAS	Parallel Sparse BLAS
RAD	Restricted Additive Schwarz
Reclamation	Bureau of Reclamation
RHS	Right-hand side
R/W	Read/Write
SIMD	Single-Instruction Multi-Data
SPD	Symmetric Positive Definite
SSOR	Symmetric Successive Over-Relaxation
TDMA	Tridiagonal Matrix

TVD	Total-Variation Diminishing
SpMV	Sparse-Matrix Vector Multiplication

Contents

Page

Executive Summary

Reclamation's mission is designed around delivering usable agricultural water for the West. Therefore, being able to simulate salinity in a 2-D finite-volume model in a realistic timeframe would be of significant value to Reclamation. The goal of this project was to develop a salinity module to simulate salinity using SRH-2D, a two-dimensional (2D) flow hydraulic and mobile-bed sediment transport model for river systems that is frequently used by Reclamation. Additionally, this research sought to improve the efficiency of SRH-2D using graphics processing unit (GPU) acceleration methods. Unfortunately, due to issues with solution stability, neither goal was achieved during the timeframe of this project, but the research laid the groundwork for future research in this area.

1. Introduction

1.1 Author's Note

The Principal Investigator of this project, Zackary Leady, left Reclamation before completing this report. As a result, this report has been compiled in his absence using a limited set of available documentation. The majority of the report consists of a literature review, as very little information is available regarding the research methodology or results.

1.2 Research Goals

Salinity is a key component of water quality. However, Reclamation currently does not have the ability to perform accurate modeling of salinity in many waterways. The goal of this project was to develop a salinity module to simulate salinity which could be coupled with or implemented into the Reclamation model SRH-2D. As described by Lai (2008), SRH-2D is a two-dimensional (2D) flow hydraulic and mobile-bed sediment transport model for river systems and has been widely used by Reclamation and outside institutions. Additionally, this research sought to improve the efficiency of SRH-2D using graphics processing unit (GPU) acceleration methods.

One intended application of this research is to the Sacramento-San Joaquin River Delta (Delta) in California. Reclamation currently uses Delta Simulation Model 2 (DSM2), a one-dimensional (1D) model, to model salinity and other water quality parameters in the Delta. The use of a 2D model such as SRH-2D with a salinity module has the potential to increase the accuracy of modeling. A graphical overview of the proposed research process is shown in Figure 1.

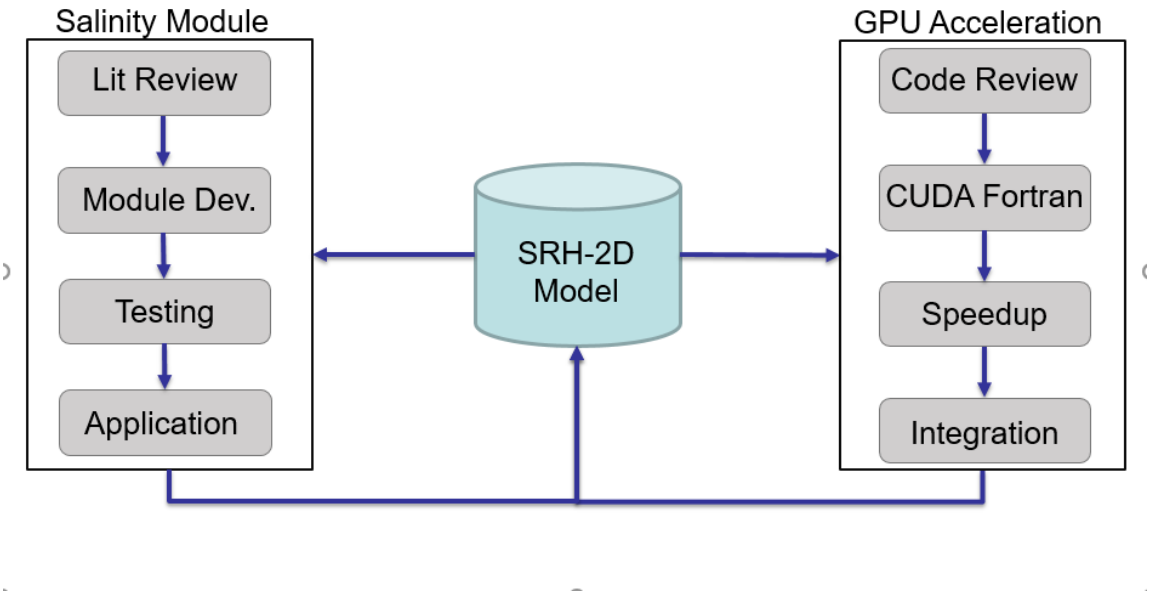


Figure 1. A graphical overview of the proposed research process.

2. Literature Review

A literature review was undertaken to better understand current knowledge regarding GPU acceleration and transport modeling. Section 2.1 summarizes the reviewed literature on GPU acceleration and its application to computational fluid dynamics problems. Section 2.2 summarizes the reviewed literature on transport modeling, including SRH-2D, and previous applications to the San Francisco Bay-Delta.

2.1 GPU Acceleration

2.1.1 Overview of GPU Acceleration

It is possible to speed up the solution process for a variety of numerical solution techniques using a GPU in place of a central processing unit (CPU), which is known as GPU acceleration. GPUs can be regarded as massively parallel vector processors (Govindaraju, 2006). Miller at al. (2013) recognized the increasing use of GPUs as a clear hardware trend in numerical simulations of water resources problems.

Modern GPUs were introduced in 1995, with general-purpose programming capabilities becoming available in the early 2000s, and the programmability increasing over time. Modern GPUs contain a mixture of 32-bit and 64-bit capability.

GPUs are designed to perform vector computations on input data represented as 2D arrays or textures. Each element of a texture is composed of four color components, and each component can store one floating point value. In order to perform computations on a data element, a quadrilateral covering the element location is rasterized on the screen. The rasterization process generates a fragment for each covered element on the screen and a user-specified program is run for each generated fragment. Since each fragment is evaluated independently, the program is run in parallel on several fragments using an array of fragment processors (Govindaraju, 2006).

2.1.1.1 GPU Streaming Model

Unlike in CPU programming, where programmers can write to any location in memory at any point in their program, GPU programming memory access is more structured. In the streaming model, programs are expressed as series of operations on data streams. The elements in a stream (that is, an ordered array of data) are processed by the instructions in a kernel (that is, a small program). A kernel operates on each element of a stream and writes the results to an output stream. These stream programming model restrictions are what allow GPUs to execute kernels in parallel (LeFohn et al., 2005).

For maximum efficiency, it is important to design an algorithm with the GPU streaming model in mind. GPUs such as the GeForce FX are characterized by inexpensive gather operations, lack of a scatter operation, and single-instruction multi-data (SIMD) semantics, which together characterize the abstract streaming model. The GPU streaming model is a specific type of abstract streaming model. Some design principles for peak performance under this streaming model are as follows:

- SIMD: Let *p* be the number of parallel pipelines; then every instruction operates on a tuple of *p* neighboring pixels. For peak performance, useful work must be made of every pixel in the tuple.
- Triangle rasterization: GPUs are most efficient at rendering triangles. Rectangles may be considered to be comprised of a pair of axis-aligned right triangles, but some work along the hypotenuse of the triangles is wasted. This waste should be minimized.
- Round-Robin Pipelining of Triangle: Streaming processors are typically multi-threaded to hide memory access latency, in which *q* independent streams are processed in an interleaved manner. The designer should optimize the value of *q* to hide memory latency while minimizing the time wasted by waiting for data-dependent instructions (Bolz et al., 2003).

The use of Compute Unified Device Architecture (CUDA) abstracts the GPU streaming model somewhat, so that programmers access memory in a way that is more comparable to a CPU. CUDA-enabled GPUs provide five different types of memory: register, shared, local, global, and constant memory.

On devices with compute capability 1.x, there are two locations where memory can reside: cache memory and device memory. For these devices, shared memory and constant cache memory are stored in cache memory. On devices that support compute capability 2.x, there is an additional memory bank that is stored with each streaming multiprocessor, which has a relatively small address space but very low access latency (van Oosten, 2011).

The organization of these different memory types is shown in Figure 2. The properties of each memory type are summarized in Table 1.

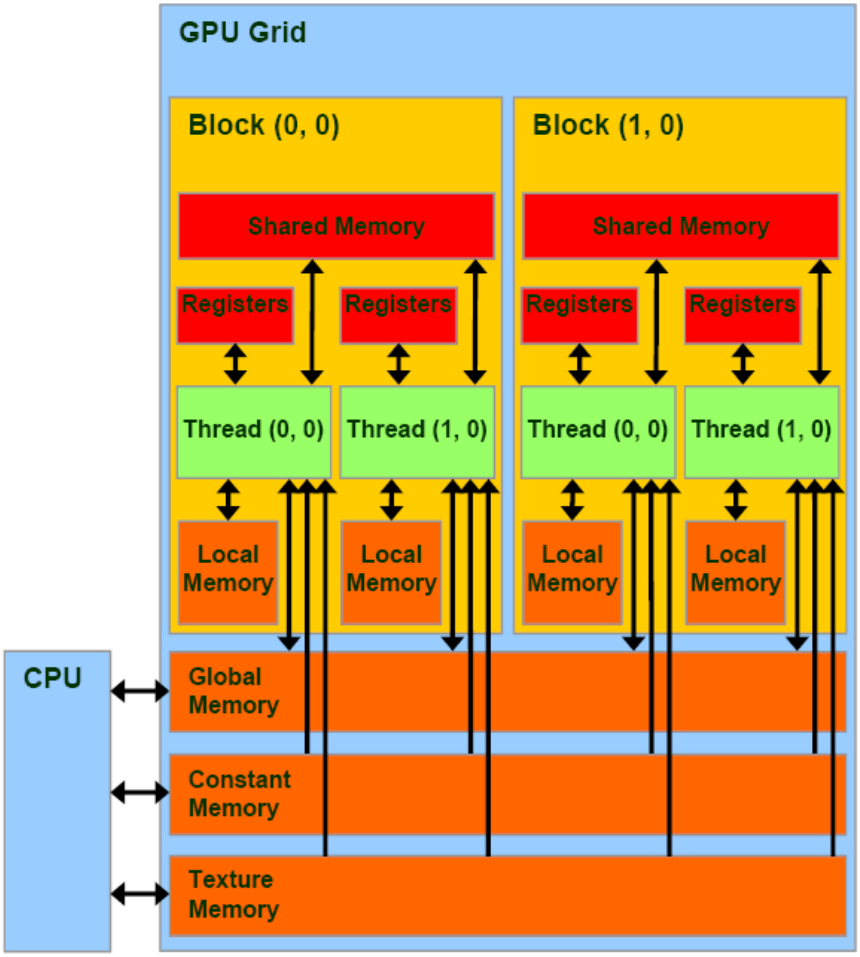


Figure 2: CUDA memory model. Figure taken from van Oosten (2011).

Table 1: Property of CUDA memory types. From van Oosten (2011).

Memory	Located	Cached	Access	Scope	Lifetime
Register	cache	n/a	Host: None Kernel: R/W	thread	thread
Local	device	1.x: No 2.x: Yes	Host: None Kernel: R/W	thread	thread
Shared	cache	n/a	Host: None Kernel: R/W	block	block
Global	device	1.x: No 2.x: Yes	Host: R/W Kernel: R/W	application	application
Constant	device	Yes	Host: R/W Kernel: R	application	application

n/a: Not applicable
R/W: Read/Write

Threads on the GPU are organized in warps, defined as a group of 32 threads of consecutive thread IDs. The first or second half of a warp is referred to as a half-warp. Global memory bandwidth can

be used most efficiently by coalescing the simultaneous memory accesses by threads in a half-warp into a single memory transaction (Li and Saad, 2010).

2.1.1.2 Gather and scatter operations

Gather and scatter operations require special consideration on the GPU. Gather and scatter are fundamental data parallel operations, in which a large number of data items are read (gathered) from or are written (scattered) to given locations. A naïve implementation can reduce performance significantly due to a low utilization of the memory bandwidth and a long,

He et al. (2007) implemented various common gather and scatter operations on an NVIDIA GeForce 8800 GPU (G80) using CUDA and tested the performance. A basic implementation of the scatter is to sequentially scan the array for scatter L and the input R_{in} once, and output all R_{in} tuples to the output R_{out} once. Likewise, the basic implementation of the gather is to scan L once, read the *Rin* tuples according to L, and write the tuples to R_{out} sequentially. However, if L is random, the scatter and gather suffer from the random access, which has low cache locality and results in low bandwidth utilization.

He et al. (2007) applied a multi-pass optimization technique to both the scatter and gather operations. This technique divides R_{out} into *nChunk* chunks, then performs the scatter in *nChunk* passes. In the *ith* pass, it scans L once, then outputs the R_{in} tuples belonging to the *ith* chunk of R_{out}, which achieves better cache locality than a single-pass scatter. This algorithm improves the scatter performance up to three times. Overall, the optimized GPU implementations are 2-7X faster than their optimized CPU counterparts.

2.1.2 Solving Sparse Linear Systems on the GPU

Sparse linear systems can be more readily parallelized than dense systems, because the solutions for certain elements are not necessarily dependent on the solutions of previous elements in the way that they are for dense systems. There are two main methods for exploiting this property of sparse linear systems for parallelization. The first is to preprocess the matrix to analyze the sparsity pattern and use the computed pattern to exploit available parallelism. The second strategy is to express the triangular matrix as a series of sparse factors. Variations of this strategy have also been applied to parallelize dense and banded triangular linear systems (Naumov, 2011).

2.1.2.1 Preconditioners

Preconditioners are algorithms that transform a linear system into a form that allows it to be solved more efficiently by numerical methods. Without preconditioning, the convergence of iterative solvers can be too slow for practical purposes (Bertaccini and Fillipone, 2016). Various types of preconditioners have been implemented on GPUs, including Incomplete LU (ILU), Incomplete Cholesky (IC), Schwarz, and approximate inverse preconditioners.

2.1.2.1.1 Incomplete LU (ILU), Incomplete Cholesky (IC), and Schwarz preconditioners

An incomplete LU factorization seeks triangular matrices L and U such that $A \approx LU$, with the result that solving for $LUx = b$ can be done quickly but does not yield the exact solution to $Ax = b$. Since the preconditioning matrix $M = LU$ and $LU \approx A$, the preconditioning operation is a back substitution followed by a forward substitution, i.e., $u = M^{-1}v = U^{-1}L^{-1}v$. If no fill-in is allowed in the ILU process, we obtain the ILU(0) preconditioner, which has the same sparsity pattern as A. The

ILU(k) preconditioner allows more fill-ins using the notion of level fill-in, and is more accurate than ILU(0). ILUT uses an alternative method to drop fill-ins, dropping them based on the numerical value of the fill-in elements; zeroing out elements smaller than a threshold (Li and Saad, 2010).

Incomplete Cholesky (IC) preconditioners are commonly used for symmetric positive definite (SPD) matrices. However, the IC factorization does not exist for all SPD matrices, and so in some cases the modified IC (MIC) factorization (Robert, 1982), which exists for any SPD matrix, must be used (Chow and Patel, 2015).

ILU factorization has been very useful in sequential computations, but is not readily parallelizable, so the basic ILU method tends to underperform on the GPU compared to the CPU, and also compared to more parallelizable algorithms on the GPU (Chow and Patel, 2015; Bertaccini and Fillipone, 2016). In many cases, it is even faster to transfer data from the GPU to the CPU and back to perform ILU factorization (Li and Saad, 2010).

One method of enhancing parallelization with ILU preconditioners is to use multicolor ordering to reorder the rows and columns of the matrix. The nodes corresponding to the graph of the matrix are colored such that no two adjacent nodes share the same color, then the matrix is reordered such that like colors are ordered together. Nodes corresponding to the same color can then be eliminated in parallel. While this method leads to more parallelism, it has the disadvantages of leading to suboptimal ILU factorizations compared to other ordering methods, and of taking away the ability to reorder a matrix to enhance solver convergence (Chow and Patel, 2015). However, in an experiment by Li and Saad (2010), a multicolor ILU(0) preconditioner achieved approximately 5X speedup of a triangular solver compared to the CPU.

Heuveline et al. (2011) developed a variation on the multicolored ILU(k) preconditioner in which they anticipate the fill-in pattern of the ILU(k) scheme, which they call the power(q)-pattern method. This modified ILU(k) method applied to a multi-colored matrix has no fill-ins in its diagonal blocks, leading to an inherently parallel execution of triangular ILU(k) sweeps. This method was implemented in the freely available finite element software package HiFlow.

The power(q)-pattern method is based on the observation that the non-zero pattern of ILU(k) grows like $|A|^{k+1}$, and thus the non-zero pattern of the factorization can be restricted by determining the pattern of $|A|^{k+1}$ to avoid dynamic memory allocation. This restriction allows control over the sparsity patterns of the L and U matrices (Heuveline et al., 2011).

Heuveline et al. (2011) tested the power(q)-pattern method on a dual-socket Intel Xeon E5450 quad-core system (eight cores in total) with an NVIDIA Tesla S1070 GPU system with four GPUs attached pairwise by PCIe to one socket each. The memory capacity of a single CPU and GPU device was 16GB and 4GB, respectively. They applied the preconditioning method to a Conjugate Gradient (CG) solver for three test matrices. The CG solver on a power(q)-pattern method preconditioned matrix is several times faster than the non-preconditioned matrix, whether they are using a sequential CPU, parallelized CPU (with openMP), or GPU. For two of the three matrices tested, the power(q)-pattern method is moderately faster than the symmetric Gauss-Sneidel and multi-colored ILU(0) methods; for the third matrix, the multi-colored ILU(0) method is fastest.

Another approach for parallelizing ILU factorizations is to split the matrix graph into various subdomains, which leads to coarse-grained parallelization. The ILU is computed in parallel for each

subdomain (Chow and Patel, 2015). There are multiple variations on this technique, depending on the method used for splitting the matrix. In the block Jacobi method, also known as the Block ILU method, the subdomains have no overlap, so one CUDA thread block can be assigned to each local block and no global synchronization or communication is needed, providing high parallelism. This method is also easy to program. However, this method usually results in a large number of iterations to converge, which can outweigh the benefits of increased parallelism (Li and Saad, 2010).

A decoupled block ILU(k) method was implemented on the GPU by Yang et al. (2017). This method was applied to reservoir simulations, where block-wise matrices appear frequently. The matrix *A* is first abstracted into a point-wise matrix *Ap*. All nonzero blocks (blocks with at least one nonzero element) in *A* become nonzero elements in *Ap*, then the fill-in nonzero pattern *P'* is established on *Ap*, and the block-wise matrix *A'* is created according to the pattern *P'*. Thus, *A'* has the same values as *A* but a different nonzero pattern, and can be used for a block-wise ILU(0) factorization directly.

This preconditioning method was tested on a GMRES solver using an Intel Xeon E5-2680 0 CPU @2.70GHz and a NVIDIA Tesla K20Xm GPU with 249.6 GB/s memory bandwidth and 2688 CUDA cores. The OS is Red Hat Enterprise Linux Server release 6.6 (Santiago) and the development environment is CUDA 6.5 and GCC 4.4. Three matrices were used for testing. Relative to the CPU, speedups for the first two matrices ranged from 1.6 to 9.4, depending on block size and k level, with smaller block sizes and smaller k-levels generally increasing speedup. For the third matrix, speedups ranged from 1.2 to 6.8 with block sizes of 1, 2, and 4, but from 0.2 to 1.2 for a block size of 8, indicating that the parallel performance is exhausted by an overly large block size (Yang et al., 2017). These experiments demonstrate the importance of optimizing both block size and k-level.

The restricted additive Schwarz method (RAS), developed by Cai and Sarkis (1999), is another method of splitting the matrix into subdomains, and is a cheaper variation on the classical additive Schwarz (AS) method. The AS method uses overlapping domains, so communication between the parallel threads is needed. With the RAS method, however, the domains have minimum overlap, making the communication cost cheaper. Cai et al. (1998) found that this method also requires fewer iterations to convergence than the AS method.

Liu et al. (2014) implemented the RAS on the GPU and tested it on various matrices, using either ILU(0) or ILUT to factor each subdomain. The preconditioned matrices were solved using the GMRES solver. They performed the test using a workstation with Intel Xeon X5570 CPU and NVIDIA Tesla C2050/C2070 GPUs. The operating system was Fedora 13 X86-64 with CUDA Toolit and GCC 4.4. The maximum speedup achieved relative to the CPU was about 10, with an average speedup of 7.8 for the ILU(0) factorization and 6.2 for the ILUT factorization.

2.1.2.1.2 Approximate inverse preconditioners

Approximate inverse preconditioning has been popular over the last two decades. This method preconditions the system $\mathbf{A}x = b$ by a direct approximation of $\mathbf{A}^{-1}$. Approximate inverse preconditioning is implemented using sparse matrix by vector multiplications, which can be implemented efficiently on highly parallel computing architectures such as the GPU (Bertaccini and

Fillipone, 2016). A disadvantage of approximate inverses is that it is difficult to predict whether the resulting matrix will be singular (Helfenstein and Koko, 2012).

Various methods for computing sparse approximate inverses have been proposed in the literature, including minimization of the residual norm, approximation by a matrix polynomial, inexact inversion of sparse triangular factors, and incomplete biconjugation. Bertaccini and Fillipone (2016) implemented the last two of these methods (inexact inversion and incomplete biconjugation) on the GPU and tested the performance.

For the inexact inversion, they followed the strategy of van Duin (1999). For effective preconditioning, a "drop strategy" is necessary to preserve the matrix sparsity. The "drop strategy" they used is based on level of fill and is called positional fill level triangular inverse, or INVK. Each iteration of the main factorization loop consists of three phases:

1. A copy-in phase, where the ith row of matrix A is expanded into a full row w
2. A factorization loop where the needed updates from the previous phase and the first dropping rule are applied
3. A copy-out phase in which the second dropping rule is applied.

The first dropping rule compares the w_k element with a user-specified threshold. For the second rule, first each element is compared with the threshold, then the p elements with the largest absolute values among those which were not dropped are retained. In order to efficiently select and remove the lowest ranked elements from a set and add elements to the set, in both the factorization and inversion phases, a partially ordered set abstract data type is used, which guarantees that both the insertion of a new element and deletion of the lowest ranked element can be performed with a cost O(log(|S|), where |S| is the cardinality of the set S.

A variation of INVK using a numerical fill drop triangular inverse is termed INVT. Both INVK and INVT have the drawback of needing to specify multiple parameters, which leads to difficulty in tuning them in actual applications. For INVK, it is necessary to choose the level of fill in the sparse factorization and the level of additional fill in the approximate inversion phase. For INVT, four parameters must be chosen: the drop threshold ϵ and the number of additional nonzeros N for both the incomplete factorization and sparse inversion.

The second method considered by Bertaccini and Fillipone (2016) was incomplete biconjugation. Their method is termed AINV and was used in RapidCFD, a GPU implementation of OpenFOAM (Arslan, 2016). Bertaccini and Fillipone's method was proposed by Benzi and Tuma (1998) and extended in Benzi et al. (2000). Biconjugation is a similar method to incomplete factorization. Like with incomplete factorization, the AINV method may break down when the coefficient matrix is not an H matrix (Benzi et al., 1996). A modified method known as SAINV was created that will not break down for positive definite matrices. SAINV was implemented by Bertaccini and Fillipone (2016) but they did not observe any clear benefits over using AINV. SAINV has also been implemented on the GPU elsewhere (Geveler et al., 2011).

The efficiency of AINV can be improved using a left-looking algorithm, where all the updates to a vector z_i involving z_j, $j<i$ are performed in a single iteration of the outer loop. The left-looking variant groups together all the updates to a given column, and suffered less from pivot breakdown in test problems. The efficiency can also be improved by using the same partially ordered set abstract data

type used for inexact inversion. The left-looking AINV algorithm variation using the partially ordered set abstract data type is denoted LLK.

Like INVT, LLK also requires the choice of two parameters. In the case of LLK, the two parameters are the dropping threshold ϵ and the amount of fill-in p. The computational complexity bounds for INVT and INVK are of the same order, which is substantiated by numerical results. The advantage of LLK is that it is normally easier to tune the control parameters of the algorithm, but once the tuning is done, the build phase of the INVT and INVK preconditioners is often faster.

These preconditioners were studied using the Parallel Sparse BLAS (PSBLAS) library, along with the development of a number of support tools, in the context of the MLD2P4 framework, a package of multilevel preconditioners that can be plugged into the PSBLAS library, using only one MPI (message-passing interface). They were tested on an Intel Xeon E5-2670 running at 2.6 GHz, coupled with an NVIDIA K20M graphics accelerator. The GPU kernels were compiled with CUDA 6.5, and all other software components were built with the GNU compilers (C and Fortran) version 4.8.3.

They started with tests based on 2D and 3D convection-diffusion. On the CPU, INVK performs comparably to solving with no preconditioning, and worse than ILU(0) preconditioning. However, on the GPU, ILU(0) performs no better than the CPU using a Hybrid (HYB) matrix format, and much worse with a Compressed Sparse Row (CSR) format (see section 2.2.2 for descriptions of these formats). INVT and LLK both perform significantly better, with speed-ups for 1.6 to 8.2 for INVT and 1.6 to 8.0 for LLK, with higher speedups for larger matrices.

They also tested these algorithms on matrices designed from an engine simulation application, which solves the turbulent Navier-Stokes equations with arbitrary Lagrangian-Eulerian (ALE) finite volume discretization solved with the SIMPLE method. The test matrices are those for the pressure-correction equation, which is the most demanding linear system in this application. The coefficient matrices are non-symmetric but with a symmetric sparsity pattern and have no more than 19 nonzero entries per row. Speed-ups with the LLK and INVT methods are lower than in the previous case but are still around 2 (1.1 to 2.8) for both methods.

Lastly, they tested the preconditioning algorithms using a subset of matrices in the University of Florida Sparse Collection, to test the effects of renumbering the matrices using either the approximate minimum degree (AMD) algorithm (Amestoy et al., 1996, 2004) or the Gibbs, Poole, and Stockmeyer variant of reverse Cuthill-McKee numbering (Gibbs et al., 1976a,b; denoted GPS). They found that AMD renumbering is practically never beneficial, while GPS reordering produces modest speed-ups compared to no renumbering sometimes but not consistently.

The LLK algorithm is available in a PSBLAS MLD2P4 plugin called MLD-AINV.

Another method for approximate inverse preconditioning is preconditioning by a matrix polynomial. The polynomial preconditioning matrix M is defined by $M^{-1} = s(A)$, where s is some polynomial. The polynomial coefficients are determined by the Stieltjes procedure. The computations used for preconditioning are SpMV and level-1 BLAS vector computations, both of which can be implemented efficiently on the GPU (Li and Saad, 2010). Li and Saad describe the algorithm by which these preconditioners are obtained, but no implementation or testing of these preconditioners on the GPU has been located, so they are not discussed further here.

2.1.2.2 Sparse-matrix vector multiplication (SpMV)

The multiplication of a sparse matrix by a dense vector (SpMV) is widely used in many linear solvers. The SpMV kernel is well-known to be a memory bounded application, and its bandwidth usage is strongly dependent on both the input matrix and on the underlying computing platform. Techniques for implementing SpMV generally involve storing the sparse matrix in a compressed format and performs the multiplication on the compressed matrix. (Fillipone et al., 2017).

Dozens of sparse matrix storage formats have been developed, each of which can be advantageous in certain circumstances. Four sparse matrix storage formats are available in cuSPARSE: Coordinate (COO), Compressed Sparse Rows (CSR), Compressed Sparse Columns (CSC), and Hybrid (HYB) (NVIDIA, 2018a). COO, CSR, and CSC are also available in PSBLAS (Fillipone and Buttari, 2018). These formats are discussed below. The ELL matrix format was introduced in ELLPACK, which is available for purchase from Purdue University (https://www.cs.purdue.edu/ellpack/).

There is no single definition of a sparse matrix, but the most famous definition is attributed to James Wilkinson: Any matrix with enough zeros that it pays to take advantage of them. To "take advantage" of the zeros essentially means avoiding their explicit storage. However, this also means that the simple mapping between the index pair and the position of the coefficient in memory is destroyed. Therefore, all sparse matrix storage formats are devised around means of rebuilding this map using auxiliary index information. The cost of rebuilding the map can vary in the context of the operations one wants to perform (Fillipone et al., 2017).

Fillipone et al. (2017) discussed 71 matrix formats that have been proposed in recent years and evaluated the performance of 7 of these formats for SpMV on the GPU. They classified the formats according to the base sparse matrix format they extend or derive from (COO, CSR, CSC, ELL, and Diagonal (DIA)). These matrix formats are described below. They also considered hybrid approaches that use multiple formats depending on the matrix sparsity pattern or other matrix parameters, and those approaches that do not directly extend any existing formats.

The COO format is a particularly simple storage scheme. It is defined by three scalars M, N, and NZ and three arrays IA, JA, and AS. The AS array contains the non-zero coefficients, the IA and JA arrays contain the row and column indices, respectively (Fillipone et al., 2017).

The Compressed Sparse Rows (CSR) format, perhaps the most popular sparse matrix representations, explicitly stores column indices and nonzero values in two arrays JA and AS, and uses a third array of row pointers, IRP, to mark the boundaries of each row. The Compressed Sparse Columns (CSC) format is extremely similar to CSR, except that the matrix values are first grouped by column, a row index is stored for each value, and column pointers are used (Fillipone et al., 2017).

The ELL (format) in its original conception comprises two 2-dimensional arrays AS and JA with M rows and MAXNZR columns, where MAXNZR is the maximum number of nonzeros in any row. Each row of the arrays AS and JA contains the coefficients and column indices; rows shorter than MAXNZR are padded with zero coefficients and appropriate column indices. This format is well suited for matrices in which the maximum number of nonzeros per row is not much larger than the matrix, and when the regularity of the data structure allows for faster code, for example, by allowing vectorization.

The DIA format in its original conception comprises a 2-dimensional array AS containing in each column the coefficients along a diagonal of the matrix, and an integer array OFFSET that determines where each diagonal starts. The diagonals in AS are padded with zeroes as necessary. This matrix format is well suited for matrices with a diagonal structure.

The seven matrix formats tested by Fillipone et al. (2017) were selected primarily based on their availability, and are as follows: The original CSR format available in cuSPARSE, the JSR variation on CSR, the HYBrid format from cuSPARSE (a mixture of ELLPACK and COO), the SELL-P format implemented in MAGMA 1.7.0, the ELLPACK-like and Hacked ELLPACK formats from their group, and the Hacked Diagonal (HDI) format from their group. They tested the formats on 31 matrices from the University of Florida sparse matrix collection, and three matrices generated from a model 3D convection-diffusion PDE with finite difference discretization, using four test platforms, with different CPUs and GPUs.

Refer to Fillipone et al. (2017) for detailed results of their comparison. One key outcome is that for matrices that come from a partial differential equation (PDE) discretization, they found ELLPACK-like formats to perform the best, provided sufficient memory is available, unless the matrix also has a piecewise diagonal structure, in which case Hacked DIA (HDI) performs the best. They also noted that there is overhead associated with the creation of sophisticated data structures, so if the matrix is only used for a few products, it may be best to use a simple format like CSR.

The original ELLPACK format is efficient for matrices with approximately the same number of nonzeros per row. For matrices with significant variation in the number of nonzeros, several variations are available. One format that aims to reduce the memory overhead is Sliced ELLPACK (Monakov et al., 2010), abbreviated as SELL or SELL-C, which preprocesses the rows and reorders and partitions them into slices of similar length, each of which is packed separately in the ELLPACK format. Each slice is assigned to a block of threads in CUDA and thread load balancing can be achieved by assigning multiple threads to a row. Slice size can be either fixed or variable. If the slice size is variable, heuristics is used to define each slice size. With an optimal slice size, the performance can be quite good, but picking the wrong slice size can adversely affect performance.

Warped ELL (Maggioni et al., 2013) is a variation on Sliced ELLPACK based on warp granularity and local rearrangement to reduce the overhead associated with the data structure, to reduce the variability of the number of nonzeroes per row and improve the data structure efficiency without affecting the cache locality. Maggione et al. (2013) found that Warped ELL achieves a reasonable performance over Sliced ELLPACK for the considered matrices.

HDI (Barbieri et al., 2010) is a variation of the DIA format used to limit the amount of padding, by breaking the original matrix into equally size groups of rows, and then storing these groups as independent matrices in DIA format. Like the original DIA format, this format is only convenient for matrices with a natural diagonal structure, often arising from the application of finite different stencils to regular grids, and is efficient for memory bandwidth (Fillipone et al., 2017).

Li and Saad (2010) also tested the performance of various matrix formats for SpMV kernels, using a workstation with Intel Xeon E5504 Processor (4M Cache, 2.00 GHz, 8-core) and an NVIDIA TESLA C1060 GPU (240 cores, 1.3 GHz, 4GB memory) running 64-bit Linux. They used matrices from the University of Florida sparse matrix collection, and from reservoir simulations. The formats

they tested were the CSR format, the vector CSR format, the Jagged Diagonal (JAD) format, and the DIA format. The CSR and DIA formats are described above.

The vector CSR format is a variation on the CSR format to assign a half-warp (16 threads) to each row, instead of only one thread, to increase the chances of memory coalescence. This technique incurs a problem releated to computing vector dot products, so to solve this problem, each thread saves its partial result into shared memory and a parallel reduction is used to sum all partial results.

The JAD format is a generalization of the ELLPACK format, which removes the assumption on fixed-length rows. To build the JAD structure, the rows are first sorted according to the number of nonzeros per row, then the first JAD element consists of the first element of each row, the second JAD consists of the second element, etc. Only one thread works on each matrix row to exploit fine-grained parallelism.

For non-diagonally structured matrices, the JAD format generally performed the best, followed by the vector CSR format, then the CSR format. All formats achieved significant speed-ups over the serial CPU implementation. A parallel CPU implementation was faster than the CSR method in some cases, but was still slower than the JAD and vector CSR methods. For diagonally structured matrices, the DIA format significantly outperformed the JAD and Vector CSR methods.

2.1.2.3 Parallel Triangular Solvers

Lower triangular problems and upper triangular problems are commonly applied in many scientific applications, such as incomplete LU (ILU) preconditioners, domain decomposition preconditioners, and Gauss-Seidel smoothers for algebraic multigrid solvers. The algorithms for these problems are serial in nature and difficult to parallelize (Chen et al., 2016).

Naumov (2011) implemented a sparse triangular linear system solve using the CUDA parallel programming paradigm, as a set of routines in the cuSPARSE library. Naumov's algorithm is focused on the situation where the same linear system needs to be solved multiple times with a single right-hand-side, as arises in the precondition of iterative methods using ILU and Cholesky algorithms. The cuSPARSE method uses the first parallelization strategy described above, splitting the solution into an "analyze" phase (which is relatively slow but only needs to be done once) and a "solve" phase (which is faster and may be done multiple times).

The principle of the "analyze" phase is to develop a directed acyclic graph (DAG) and to traverse it using, for example, a modified breadth-first-search, which visits each node's children only if they have no dependencies on the other nodes. The purpose of the search is to group the independent nodes into levels, which are then passed to the "solve" phase.

The algorithm of the solve phase is as follows (with notation modified from the original source for consistency with that used in other papers):

1: Let *nlev* be the number of levels.
2: **for** *e* =1:*nlev* **do**
3: *list* = the sorted list of rows in level *e*.
4: **for** *row* ∈ *list* in parallel **do** > Process a Single Level

5: Compute the element of the solution corresponding to row.
6: end **for**
7: Synchronize threads. > Synchronize between Levels
8: end **for**

A comparison was made between standalone sparse triangular solvers using the cuSPARSE level scheduling on the GPU and using Multiple Kernel Learning (MKL) on the CPU, made using the hardware system with NVIDIA C2050 (ECC on) GPU and Intel Core i7 CPU 950 @ 3.07GHz, using the 64- bit Linux operating system Ubuntu 10.04 LTS, cuSPARSE library 4.0 and MKL 10.2.3.029 (Naumov, 2011).

MKL outperforms the cuSPARSE method for a single solve. However, there are many cases where the solution of the sparse triangular linear system needs to be repeated multiple times, so the time taken by "solve" phase becomes more significant than that of the "analyze" phase, which only needs to be performed once. The "solve" phase of the cuSPARSE method is faster than the MLK method 14 out of 17 times. For these matrices, the number of iterations needed to catch up with MKL's performance ranges from 4 to 80 (Naumov, 2011).

Chen et al. (2016) developed another method to speed up solutions to linear triangular systems, which included a new matrix format, denoted by HEC (Hybrid ELL and CSR). An HEC matrix contains two submatrices: an ELL matrix, which was introduced in ELLPACK, and a Compressed Sparse Row (CSR) matrix. The ELL matrix also has two submatrices: a column-indices matrix and a non-zeros matrix. The length of each row in these two matrices is the same. In the HEC format, the regular part of a given triangular matrix L is stored in the zero part, and the irregular part is stored in the CSR part.

Like the cuSPARSE level scheduler solver, Chen et al.'s solver has an "analyze" phase, which groups the nodes into levels, and a "solve" phase. The steps of the "analyze" phase are as follows:

1. Calculate the level of each unknown using the following equation:
   $l(i) = 1+\max l(j)$ for all j such that $L_{ij} \neq 0$, $i = 1,2,\ldots,n$,
   where L_{ij} is the (i, j)th entry of L, $l(i)$ is zero initially, and n is the number of rows.
2. Calculate the map $m(i)$ using the following equation:
$$m(i) = \sum_{j=1}^{k-1} N_j + p_k(x(i)), x(i) \in S_k$$
   where $p_k(x(i))$ is the position of $x(i)$ in the set S_k when $x(i)$ belongs to S_k.
3. Reorder matrix L to L' using the map $m(i)$.
4. Convert L' to the HEC format.

The lower triangular problem is then solved using the following algorithm:

1: Let nlev be the number of levels
2: **for** i=1:n **do**
3: $b'(m(i)) = b(i)$;
4: **end for**
5: **for** i = 1:nlev **do**
6: start = level(i)

```
7:      end = level(i + 1) – 1
8:      for j = start:end do
9:              solve the jth row
10:     end for
11: end for
12: for i = 1:n do
13:     x(i) = x'(m(i));
14: end for
```

In the above algorithm, *level*(*i*) is the start row position of level *i*. First, the right-hand side *b* is permutated according to the map *m*(*i*) that was computed. Then the triangular problem is solved level by level and the solution in the same level is simultaneous. At the end, the final solution is obtained by a permuatation (Chen et al., 2016).

The upper triangular problem is mapped to a lower triangular problem using the following transferring map:

$$t(i) = n - i$$

where *n* is the number of rows.

The performance was tested using a workstation with Intel Xeon X5570 CPUs and NVIDIA Tesla C02050/C2070 GPUs, with a Cent OS 6.2 X86_64 with CUDA Toolkit 4.1 and GCC 4.4.

Three preconditioners were tested on two real-world sparse matrices: block ILU(0), block ILUT, and Restricted Additive Scharwz (RAS). The solver with block ILU(0) preconditioning was sped over three times faster, with block ILUT preconditioning was sped around 2 times, and with RAS preconditioning was sped up to 7 times faster. Using an SPE10 benchmark matrix, which is highly heterogeneous and designed to be hard to solve, the average speedup was around 6 with block ILU(0) preconditioning and 5 with RAS preconditioning. Speedup was not achieved with block ILUT preconditioning for this matrix, due to the matrix's irregular non-zero pattern.

2.1.2.4 Krylov Subspace Solvers

Krylov subspace solvers are a class of linear solvers in which the *k*th iteration minimizes some measure of error over the *k*th shifted Krylov subspace

$$K_k = \text{span}(\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \ldots, \mathbf{A}^{k-1}\mathbf{r}_0)$$

for $k \geq 1$ (Miller et a., 2013). These are the most commonly used linear solvers for computational fluid dynamics problems. Three kinds of Krylov subspace solvers are discussed here: Conjugate Gradient (CG), Bi-Conjugate Gradient Stabilized (BiCGSTAB), and Generalized Minimum Residual (GMRES).

Krylov subspace solvers perform well when the coefficient matrix is close to the identity matrix or has a few small clusters of eigenvalues and is diagonalizable. Discretizations of differential operators lead to poorly conditioned linear systems, so applying a suitable preconditioner is key to making Krylov solvers perform well (Miller et al., 2013).

2.1.2.4.1 Conjugate Gradient (CG) Solvers

Conjugate gradient (CG) solvers are popular for solving linear systems on the GPU that involve a real, symmetric, and positive definite coefficient matrix and a real right-hand side, due to their

parallelizability (Michels, 2011). The conjugate gradient algorithm was introduced by Hestenes and Stiefel (1952). The kth iteration minimizes $||\mathbf{u}\text{-}\mathbf{u}^*||_A$ over $\mathbf{u}_0 - K_k$, where $\mathbf{u}^* = \mathbf{A}^{-1}\mathbf{f}$ is the solution and the A-norm is defined by:

$$||\mathbf{u} - \mathbf{u}^*||_A = \sqrt{(\mathbf{u} - \mathbf{u}^*)^T \mathbf{A}(\mathbf{u} - \mathbf{u}^*)}$$

(Miller et al., 2013). The implementation of a conjugate gradient solver involves only a few types of non-trivial operations: scaled vector additions, dot products, and matrix-vector multiplications (Bolz et al., 2003; Michels, 2011). Thus, the efficiency of the CG solver is largely controlled by the efficiency of these three operations. CG is very efficient in terms of storage, needing only five vectors for the entire iterative method, plus the storage necessary for the coefficient matrix.

An early (pre-CUDA) application of GPUs to solving sparse linear systems was done by Bolz et al. (2003), who implemented a CG solvers and a multigrid solver on the GPU and compared their performance to CPU implementations. As the multigrid solver is applied only to regular grids, it is not discussed here. The GeForce FX GPU was used for their implementation.

Implementing a conjugate gradient method or one of its variants on the GPU requires the construction of (1) data structures for sparse matrices, (2) data parallel algorithms for sparse matrix vector multiplies, and (3) reduction operators for inner product computations. If the underlying PDE is non-linear, it is desirable to compute the matrix entries on the GPU as well. The algorithm of Bolz et al. (2003) accommodates this, making it well suited for both linear and non-linear PDEs on unstructured meshes.

In a vector multiply operation, the fragment program executes the inner product between a given row and a vector of unknowns. Fragment programs must execute in lockstep, with no branching or early termination. Bolz et al. (2003) improved the efficiency of this step by "rendering" groups of rows with equal numbers of non-zero entries.

For problems of the type y = **Ax**, in the case that the entries of **A** depend on **x** (for example, when the coefficient of a given edge is controlled by the two incident triangles, which are in turn described by their incident vertices), two additional kernels are required: one to update A_i^x and another for A_j^a. In traditional FEM codes, this is typically done by iterating over all elements computing local stiffness matrices, then accumulating these local stiffness matrices into a global stiffness matrix. However, this procedure requires a scatter operation. Scatter operations were not available on current-generation GPUs (Bolz et al., 2003), but were made available in 2007 with NVIDIA's release of CUDA (Ye et al., 2007) – see the "Gather and scatter operations" section. As a result of this limitation, the non-zero entries were computed directly.

The sum reduction of vector inner products is made more efficient by rendering a quadrilateral with half the dimension along either axis, summing four elements. This process is repeated until a single pixel quadrilateral is rendered containing the results of the sum reduction. If the length of the vector is not a power of two, odd-length dimensions will result. These are dealt with by placing zeroes in unused pixels (Bolz et al., 2003).

The cuSPARSE sparse triangular solver discussed above was also analyzed in the context of solving a linear system using preconditioned Bi-Conjugate Gradient Stabilized (BiCGStab) and CG iterative methods for non-symmetric and symmetric positive definite (SPD) matrices, respectively. They were preconditioned using ILU and Cholesky factorizations, respectively. Both the pseudocode and an

implementation in the C programming language using the cuBLAS and cuSPARSE libraries is provided in Naumov (2018).

In a performance comparison using the incomplete-LU and Cholesky preconditioned iterative methods implemented on the GPU, using the cuSPARSE and cuBLAS libraries achieved an average of 2x speedup over their MKL implementation. The best speedup was obtained using incomplete-LU and Cholesky factorization with 0 fill-in. In general, the speedup for the incomplete factorizations decreased as the threshold parameters were relaxed and the factorization became more dense. These comparisons were made using the hardware system with NVIDIA C2050 (ECC on) GPU and Intel Core i7 CPU 950 @ 3.07GHz, using the 64- bit Linux operating system Ubuntu 10.04 LTS, cuSPARSE library 4.0 and MKL 10.2.3.029 (Naumov, 2011).

Scaled vector additions take the form of $sum = x + \lambda y$, where λ is the scaling factor and x and y are vectors with n components each. This addition can be realized with n threads which execute in parallel. If t threads run in parallel, the runtime can be decreased from $O(n)$ to $O(n/t)$ (Michels, 2011).

Michels (2011) implemented a CG solver using CUDA that used the ELLPACK-R matrix format and a SSOR preconditioner; however, while their algorithm achieves 4-7X speedups over an MKL implementation on the CPU, in testing they found their algorithm to be about 2X slower than the CG solver included in the CUDA Toolkit 3.2, which uses the CSR format.

Müller et al. (2014) took a different approach to implementing a CG solver on the GPU, using a matrix-free implementation. They focused on the elliptic PDE for the pressure correction arising in the dynamical core of numerical weather- and climate-prediction models. They note that similar implementations could be developed for other types of Krylov subspace iterative methods.

Their algorithm does not explicitly store either the coefficient matrix nor the preconditioner matrix. For matrices arising from the discretization of PDEs, the local matrix stencil only couples each grid to its neighbors. Because memory access is more expensive than floating point operations on GPUs, they recalculate the stencil whenever it is needed in the SpMV of preconditioner solve. An additional advantage of the matrix free method is that there are no matrix setup costs, and the costs for precomputing the vectors is negligible.

Müller et al. (2014) took an interleaved approach to preconditioning, such that the main iteration consists of only two loops over the grid, each of which contains either the SpMV or the tridiagonal solve and a number of BLAS operations. This reduces the number of memory references, especially if the cache can be used efficiently.

They tested their algorithm using the GPU node of the aquila cluster in Bath, which contains an Intel Xeon E5-2620 Sandybridge CPU with a clockspeed of 2.0 GHz and an NVIDIA Fermi M2090 GPU. The M2090 GPU contains 512 cores running at a clockspeed of 1.3 GHz with are organized into 16 streaming multiprocessors with 32 cores each. They used version 4.4.6 of the gnu C compiler for compiling the CPU code, and the NVIDIA nvcc compiler for compiling the CUDA code. The optimized BLAS and LAPACK libraries were used for the CPU code. The tested matrix had a size of 256 x 256 x 128, which is a typical size for meteorological applications (Müller et al., 2014).

The matrix-free CUDA implementation of the preconditioned CG solver is 3-5 times faster than the matrix-explicit CSR CUDA implementation, and 30-60 times faster than the CPU implementation. The interleaved version is about 1.5 times faster than the non-interleaved version (Müller et al., 2014).

Phillips and Fatica (2016) compared the performance of various high-performance GPU CG algorithms, using a preconditioner based on symmetric Gauss-Seidel smoothing (SYMGS). Their considered set of algorithms included the baseline CuSPARSE algorithm (which requires a CSR matrix format), the CUSPARSE algorithm with color ordering, custom kernels with color ordering and the CSR format, and custom kernels with color ordering and the ELL format. The use of custom kernels allows the adoption of a more flexible matrix format which simplifies the reordering of the matrix, and removes the need for sorting of the row elements with respect to the diagonal.

They tested these four versions on a K20X GPU with ECC enabled, with a 128^3 domain. The matrix reordering had the strongest improvement in performance, since it exposes more parallelism in the SYMGS preconditioning routine. Overall, the fourth version that includes custom kernels with color ordering and ELL format was the fastest, with more than 4x speedup over the baseline CuSPARSE algorithm. They then tested this optimized version on several GPU platforms, which produced the fastest results per processor ever recorded.

2.1.2.4.2 Generalized Minimum Residual (GMRES) Solvers

GMRES solvers are used for the numerical solution of nonsymmetric systems. The method was developed by Saad and Schultz (1986). The kth GMRES iteration minimizes $||\mathbf{f} - \mathbf{Au}||_2$ over $\mathbf{u}_0 - K_k$. The GMRES method requires substantially more storage than the CG method, as it must store a basis of k orthonormal vectors for the Krylov subspace, a demand which can be impossible to meet for large problems. Low-storage alternatives to GMRES have been developed, but they all have limitations. One method of limiting storage in the GMRES method is by limiting it to an m-dimensional Krylov subspace, then restarting the iteration when storage is exhausted, a method known as GMRES(m). However, GMRES(m) does not share the rigorous convergence theory of GMRES, and can fail to converge (Miller et al., 2013).

A GPU GMRES solver was implemented by Liu et al. (2015) for the solution of a finite difference-based thermal simulation algorithm. This algorithm was implemented using NVIDIA CUDA and run on a Tesla C2070 GPU, which has 488 cores of 1.15 GHz and 5-GB global memory. The CPU results were tested on a quad-core Xeon E5620 machine at 2.00 GHz with 28-GB memory. They tested three solvers: serial GMRES on CPU, parallel GMRES on a CPU-GPU platform with preconditioners, and a parallel LU solver called superLU_MT. The GPU preconditioners used were AINV and ILU0, and DIAG. The GPU implementation was up to 4X faster than the CPU implementation, and up to 700x faster than the SuperLU solver, though the speedup was highly variable and for some smaller matrices was less than 1 with AINV preconditioning, as the speedup in solving time was insufficient to outweigh the time required for preconditioning. The DIAG preconditioner performed best for small matrices, but failed to converge for larger matrices. For all matrices, speedup was higher for ILU0 preconditioning than for AINV preconditioning, though Liu et al. (2015) note that the AINV preconditioner is more accurate.

2.1.2.4.3 Bi-Conjugate Gradient Stabilized (BiCGSTAB) Solvers

BiCGSTAB is another non-symmetric CG solvers. It does not require a transpose-vector product, but it does need two matrix vector products and does not have a complete convergence theory. BiCGSTAB can fail to converge in certain situations, so the user must be prepared to reinitialize failed iterations when necessary (Miller et al., 2013).

Naumov (2018) provide both pseudo-code for the GPU implementation of the BiCGSTAB algorithm, and an implementation using the C programming language and the CuSPARSE and CuBLAS libraries. The tests described in the CG section were also performed using the BiCGSTAB algorithm. The BiCGSTAB algorithm produced average speedups of 2.1-2.7 relative to the CPU implementation.

2.1.2.5 Tridiagonal solvers

Tridiagonal matrices arise in many engineering and scientific applications, and thus the triangular solver is a critical building block for such applications. Chang and Hwu (2014) note that there is no single best tridiagonal solver for all applications, as each application may have different requirements, such as data with different layouts, matrices with a certain structure, or execution on multi-GPUs. Most tridiagonal solvers on the GPU contain two components: partitioning methods to divide workload for parallel computing, and optimization techniques for independent solvers.

cuSPARSE offers both pivoting (gtsv) and non-pivoting (gtsv_nopivot) tridiagonal solvers. The pivoting solver offers better accuracy and stability at the expense of some execution time. The non-pivoting algorithm uses a combination of the Cyclic Reduction (CR) and the Parallel Cyclic Reduction (PCR) algorithms to find the solution. It achieves optimal performance when the size of the linear system is a power of 2 (NVIDIA, 2018a).

The CR algorithm, also known as odd-even reduction, contains two phases, forward reduction and backward substitution. In every step of the forward reduction, each odd (or even) equation is eliminated using the adjacent two even (or odd) equations. After a step of CR forward reduction, redundant unknown variables and zeros can be removed, and a half-size matrix is formed of the remaining unsolved equations. Each step of the backward substitution solves for unknown variables by substituting solutions obtained from the smaller system (Chang and Hwu, 2014).

The PCR algorithm differs from CR in that is only performs the forward reduction, and the forward reduction is performed on all equations, instead of odd (or even).

Another tridiagonal solver algorithm is the SPIKE algorithm, which was originally introduced by Sameh and Kuck (1978) and modified by Pollizi and Sameh (2006). It is a domain decomposition algorithm, that partitions a matrix into block rows containing diagonal sub-matrices and off-diagonal elements (Chang and Hwu, 2014). The SPIKE algorithm source code is provided by Chang et al. (2012) at http://impact.crhc.illinois.edu.

Chang and Hwu (2014) developed a new hybrid algorithm, SPIKE-CR, as a case study to demonstrate how to apply optimization techniques. In the SPIKE-CR, the SPIKE algorithm is applied to partitioning, due to its lower computation overhead than PCR and lower memory access overhead than other domain partitioning methods. CR is then applied for the independent solver. Various optimization techniques are applied, including register packing, CR/PCR hybridization, a new warp level PCR, and another level partitioning using the SPIKE algorithm to minimize

communication between warps within a thread block. Some code fragments for this method are provided in Chang and Hwu (2014).

Chang and Hwu (2014) compared their new SPIKE-CR method to the SPIKE method and the cuSPARSE non-pivoting method. The SPIKE-CR method produced a speedup of 1.2 over the SPIKE method, and a speedup of 2.2 over the cuSPARSE non-pivoting method, since SPIKE-CR has no data marshaling overhead and less memory access overhead.

2.1.2.6 Summary of solution methods for sparse linear systems

To obtain a fast, accurate, and stable solution to sparse linear systems, developers must choose a preconditioning method (or choose not to precondition matrices), matrix storage format, and solver. Consideration of preconditioning methods and matrix storage formats for application to this project was limited to methods that have been implemented on the GPU and whose implementation is readily available for use. This primarily consists of the methods available in cuSPARSE.

The two preconditioners available in cuSPARSE are ILU(0) and IC preconditioners. Despite the proliferation of efficient preconditioners in recent years, these two preconditioners remain popular. While, as mentioned in section 2.1.2.1.1, the ILU(0) algorithm is not readily parallelizable, the cuSPARSE implementation of ILU(0) computes levels to extract more parallelism from both the ILU(0) and IC preconditioners (NVIDIA, 2018a). As a result, IC preconditioning is a strong candidate for SPD matrices, and ILU(0) for non-SPD matrices.

cuSPARSE supports the following matrix formats: COO, CSR, CSC, and HYB. There are also two variations on CSR available: Block CSR (BSR), and Extended BSR (BSRX), which are appropriate for sparse matrices with dense submatrices (NVIDIA, 2018a). The HYB matrix is a hybrid of COO and ELL matrix formats. Given various format comparisons discussed above, the HYB format is likely to outperform the other available formats. However, a simpler matrix format like COO, CSR, or CSC may perform better if the matrix is only used for a few operations, given the time required to convert a matrix into a complex format.

The CG and BiCGStab solution methods are both popular for solving sparse linear systems on the GPU, due to their parallelizability, and implementations of both methods using the cuSPARSE and cuBLAS methods are available (NVIDIA, 2018b). The primary advantage of the BiCGStab method is its applicability to non-symmetric matrices. For SPD matrices, the CG method is generally preferable, due to its more complete convergence theory.

2.1.3 Applications of GPU Acceleration to Computational Fluid Dynamics Problems

Amouzgar et al. (2016) implemented a Tsunami model, using a second-order accurate hydrodynamic model, on the GPU using the CUDA framework and achieved speed-ups of 45 to 64 relative the CPU. Their model solves the 2D shallow water equations using a finite volume Godunov-type scheme incorporated with an HLCC approximate Riemann solver.

Ha et al. (2018) applied GPU acceleration to the solution of the incompressible Navier-Stokes equations using a fractional step method. Ha et al. note that fully explicit schemes for integrating the Navier-Stokes equations are the most readily parallelizable, as the problem can be decomposed into tasks operating on independent data sets. However, semi-implicit schemes, such as the fractional step method, are more commonly used for solutions of wall-bounded incompressible flows. This

method integrates the convective terms explicitly and the viscous terms implicitly. The implicit treatment of the viscous terms allows a stable solution even at a larger time-step size.

When the viscous terms are discretized using a second-order central-difference scheme and are integrated implicitly, the resulting momentum equations require inversion of multiple tridiagonal matrices (TDMAs). Tridiagonal matrices can easily be inverted using the Thomas algorithm, but this algorithm is inherently difficult to parallelize (Ha et al., 2018).

Each of the Navier-Stokes solver comprises three major sub-steps: the computation right-hand side (RHS) of momentum equations, the alternating direction implicit (ADI) solver used for solving for the velocity, and the Poisson equation solution, which is transformed with a Fourier transform then solved directly. In a traditional single-core CPU implementation, the ADI solver takes the most time (Ha et al., 2018).

The RHS computation involves many arithmetic operations arising from finite differences. The computation can be expressed as a triply nested loop, which can easily be parallelized as CUDA kernels. The simplest way to map the RHS is to CUDA kernels is to use CUDA Fortran compiler directives, which instruct the compiler to automatically generate asynchronous kernels from the host code containing tightly nested loops. Alternatively, one can transform the triply nested loop into a kernel by mapping loop indices onto thread and block indices, which is moderately faster. The RHS kernel can be further optimized by using a cache configuration preferring L1, and by substituting locally defined temporary variables of the kernel into shared memory variables (Ha et al., 2018).

The ADI solver requires six inversions of general TDMAs and three inversions of periodic TDMAs at each sub-step. The TDMA solver is parallelized using the hybrid CR-PCR tridiagonal method implemented in cuSPARSE, discussed previously. At boundaries, periodic TDMAs arise, which is primarily a TDMA with a few additional nonzeros. This problem can be converted into the inversion of two TDMAs using the Sherman-Morrison Algorithm (Ha et al., 2018).

The parallelism of the ADI solver can be further improved using multi-level parallelism. In 2-level parallelism, the first level is the equation-level at which equations are eliminated in parallel using reduction algorithms, and the second level is the matrix-level at which multiple TDMAs of one coordinate direction are inverted in parallel, which can be achieved in cuSPARSE using cusparseDgtsvStridedBatch to invert multiple matrices concurrently. In 3-level parallelism, the first and second levels are the same as in 2-level parallelism. The third level is the velocity-level at which multiple TDMAs are inverted together in parallel. 3-level parallelism inverts the matrices for all u_1-, u_2-, and u_3-momentum equations in parallel. A fourth level can be used to maximize the workload for GPUs and minimize dynamics allocation, however, 4-level parallelism has issues regarding memory capacity (Ha et al., 2018).

The Poisson equation solution requires a half-range cosine transform in the x_1 direction and a Fourier transform in the x_3 direction, both of which can be computed with a Fast Fourier Transform (FFT), which are computed using functions from the cuFFT library. The second-order central discretization results in multiple TDMAs, the inversion of which is the bottleneck for GPU acceleration. This linear system has real-valued diagonals on the left-hand side and a complex-valued right-hand side. The TDMAs must be inverted once for the real part of the right-hand side, and another time for the imaginary part, using an algorithm similar to the Sherman-Morrison algorithm used for the ADI solver. Inversion of the TDMAs in the Poisson equation are unstable and

therefore require pivoting (Ha et al., 2018). As discussed above, the cuSPARSE pivoting algorithm is slower but more accurate and stable compared to the non-pivoting algorithm.

Numerical experiments were conducted to compare the GPU code with a highly-optimized single-core CPU counterpart. The CPU code was run on a CentOS 6.5 Linux server with two deca-core Xeon E5-2660 v3 @2.6 GHz CPUs, and was compiled with an Intel Fortran Compiler v. 16.0.3. The GPU code was run on a CentOS 6.8 workstation with an Xeon E5-2630 v3 @2.4 GHz CPU along with an NVIDIA Tesla K40c GPU, and compiled with a PGI Fortran Compiler v. 16.1.0. Additional performance tests of the GPU solver are conducted on a modern GPU server, IBM Power System S822LC for High Performance Computing, which is equipped with two octa-core Power8 CPUs and four Tesla P100 GPUs, but only a single GPU was utilized for this study. The GPU code was run on Ubuntu 16.04 and was compiled with a PGI Fortran Compiler version 17.4 (Ha et al., 2018).

The maximum grid size supported on the GPU was estimated to be 134 million on the Tesla K40C and 190 million on the Tesla P100. Speed-ups relative to the CPU for different grid sizes range from 5.7 to 20.0 on the Tesla K40C and 6.4 to 45.4 on the Tesla P100, with greater speedups for large grid sizes (note that the Tesla K40c was not used for the largest three grids due to its memory limitations). The Tesla P100 has more than 3 times higher computational power than K40c due to an increase in the number of DP cores and core frequency, but the solver runs only 2.4 times faster on the Tesla P100 than on the Tesla K40c, due to memory bandwidth limitations (Ha et al., 2018).

Ha et al. (2018) also compared their ADI solve method to a preconditioned conjugate gradient (PCG) method. The ADI method is generally faster from the viewpoint of operation counts, but has the drawbacks of difficulties in parallelization of TDMA inversion and in multiple data transfers. For the comparison, Ha et al. implemented a conjugate gradient (CG) method using built-in functions from cuBLAS and cuSPARSE – one case without a preconditioner, and one with an ILU(0) preconditioner, following the code provided in NVIDIA (2018b). The CG method is fastest for grids with between 4 and 50 million cells, while the ADI method is faster for larger grids.

Helfenstein and Koko (2012) also developed a method for solving the Poisson equation on the GPU. They used the PCG method with a Symmetric Successive Over-Relaxation (SSOR) approximate inverse preconditioner. They tested this method using an Intel Xeon Quad-Core CPU with 2.66 GHz, 12 GB RAM using gFortran, and an NVIDIA Tesla T10 GPU with 240 cores and 4 GB RAM using CUDA. For matrix sizes ranging from 265,000 to 2.1 million, using 8-thread warp per row, they achieved speedup of their PCG method ranging from 1.2 to 1.9 relative to the CG method on the GPU (less speedup for larger matrices) and 6.2 to 10.3 relative to the CG method on the GPU (more speedup for larger matrices). They also tested the PCG algorithm using a different implementation of SpMV that splits multiplication and additional operations, which is faster than the basic SpMV implementation only for the smallest matrices tested.

Kao and Sheu (2018) developed a finite element solver on multiple GPU cards for solving three-dimensional incompressible Navier-Stokes equations. They discretized the Navier-Stokes equations using the streamline upwind finite element model. In this finite element flow solver, all the elementary matrices can be only stored in an element level; there is no need to assemble a global matrix, which reduces the amount of computer memory needed. To get an unconditionally convergent solution, they transformed the asymmetric and indefinite matrix equations into an equivalent SPD counterpart by multiplying its transpose on it. Since the matrix equations are then

SPD, the CG iterative solver can be applied to get an unconditionally convergent solution. However, the use of this approach increases the condition number and makes the convergence of CG very slow. The convergence can be accelerated with the use of a suitable preconditioner. They selected the Jacobi preconditioner due to its easy parallel implementation.

The matrix-vector product is the most expensive operation in the algorithm developed by Kao and Sheu (2018). Since the global matrix is never assembled, they decomposed the matrix-vector product into a sum of element-level matrix-vector products, using a mesh coloring technique to ensure that any two elements in a given subset do not share the same node. Their finite element code was written in CUDA Fortran and compiled with the PGI accelerator. They used an Intel E5-2690 V4 CPU with 14 cores, 1.54 TB off-chop memory, 37.1 GB/s peak flops, and 76.8 GFlops/s memory bandwidth, coupled with an Nvidia Pascal P100 GPU with 3584 single-precision cores (1792 double-precision), 16 GB off-chip memory, 10.6 TFlops/s peak flops (single-precision), 5.3 TFlops/s (double-precision), and 732 GB/s memory bandwidth. They tested the algorithm using a three-dimensional lid-driven cavity flow problem, with problem sizes ranging from 1.6 million to 10.8 million degrees of freedom. The speed-up relative to the CPU ranged from 56.4 to 63.9 for a single GPU, 66.3 to 117.0 for two GPUs, and 74.9 to 134.4 for four GPUs, with larger speed-ups for larger problems. Note that the largest problems were run only using 4 GPUs.

Liu et al. (2018) solved the 2D shallow water equations based on an unstructured Godunov-type explicit finite volume scheme for flood simulation, termed the Monotone Upstream Scheme for Conservation (MUSCL)-Hancock scheme, with triangular computational grids. Rather than using CUDA, they used the OpenACC programming interface, which is collection of runtime routines and compiler directives that use FORTRAN of C/C++ languages to compile the specified code blocks of computational loops. Due to the explicit nature of their scheme, they were able to take advantage of the natural parallelism in their independent data loops.

Liu et al. (2018) tested their model on an Intel Xeon E5-2690 CPU @ 3.0 GHz with a Tesla K20 card with a Kepler GK110 GPU and 2496 NVIDIA CUDA cores. They first used two simple test cases for validation, then applied it to a real-world application with three grid-division schemes, ranging from 179,000 to 2.9 million triangular elements, and simulated a dike breach. The speedup on the GPU relative to the CPU ranged from 11.3 to 31.1, with higher speedups for larger grid sizes.

Tomczak et al. (2013) provided another application of GPU acceleration to the numerical solutions of the Navier-Stokes equations. They analyzed the pressure implicit with operator splitting (PISO) and semi-implicit method for pressure linked equation (SIMPLE) solvers on unstructured grids. Their GPU implementation of PISO and SIMPLE followed the CPU implementation of Jasak (1996) and Weller et al. (1998). Their implementation uses Jacobi preconditioning and the ELL matrix format.

They tested their algorithm on the Tesla C2070 GPU attached to a CPU running 64-bit Ubuntu 10.04 LTS, graphics driver v.290.10, CUDA 4.1, and gcc 4.4.3. The reference CPU tests were performed using OpenFOAM v.1.7 on a dual-socket Intel Xeon X5670 processor system running 12 MPI processes to fully saturate all available CPU cores. CPU tests were performed using both the simple Jacobi preconditioning used on the GPU, and the geometric-algebraic multi-grid solver (GAMG), considered to be among the fastest solvers of the pressure equation available in OpenFOAM. They solved three different CFD problems: steady flow in a 3D lid-driven cavity, the

transient Poiseuille flow in two dimensions, and the steady flow through the human left coronary artery, with regular mesh resolutions varying from 10^3 to 223^3 cells.

The GPU solver is slower than the CPU for meshes of less than approximately 10^5 cells, and significantly faster when the mess has more than 10^6 cells, up to 4.2x. However, the GPU solver is slower than the CPU with GAMG preconditioning for most of the cases considered, due to the higher number of iterations necessary to achieve convergence with Jacobi preconditioning (Tomczak et al., 2013), illustrating the importance of selecting an optimal preconditioner.

2.2 Transport Modeling

2.2.1 Transport Modeling Overview

Water quality models depend on the principle of mass balance. Within a segment of a water quality model, the components of mass balance include changes by transport into and out of the segment, changes by physical or chemical processes occurring within the segment, and changes by sources or discharges to and from the segment. Changes by transport include both advective transport (transport by flowing water) and dispersive transport (transport resulting from concentration differences). Advective transport generally dominates in flowing rivers, while dispersion is the predominant transport mechanism in estuaries subject to tidal action (Loucks and van Beek, 2005).

The advective transport, $T_{x_0}^A$, at site x_0 is the product of the average water velocity at that site, v_{x_0}, the surface or cross-sectional area, A through which advection takes place, and the average concentration of the constituent, C_{x_0}:

$$T_{x_0}^A = -v_{x_0} \times A \times C_{x_0} \tag{1}$$

The dispersive transport, $T_{x_0}^D$, across a surface area A can be calculated as:

$$T_{x_0}^D = -D_{x_0} \times A \times \frac{\partial C}{\partial x}|_{x=x_0} \tag{2}$$

where D_{x_0} is the dispersion or diffusion coefficient at site x_0 and $\frac{\partial C}{\partial x}|_{x=x_0}$ is the concentration gradient at site x_0 (Loucks and van Beek, 2005).

In one dimension, the principle of mass balance results in the following advection-diffusion equation, also known as the generic scalar transport equation (valid only for passive scalar transport without source/sink terms):

$$\frac{\partial C}{\partial t} = \frac{\partial}{\partial x}\left(D\frac{\partial C}{\partial x}\right) - \frac{\partial}{\partial x}(vC) \tag{3}$$

where C is the average concentration, D is the dispersion of diffusion coefficient, and v is the average velocity. In three dimensions, this equation becomes:

$$\frac{\partial C}{\partial t} = D_x\frac{\partial^2 C}{\partial x^2} - v_x\frac{\partial C}{\partial x} + D_y\frac{\partial^2 C}{\partial y^2} - v_y\frac{\partial C}{\partial y} + D_z\frac{\partial^2 C}{\partial z^2} - v_z\frac{\partial C}{\partial z} \tag{4}$$

with dispersion coefficients D_j defined for each direction. The advection-diffusion reaction equation emerges by adding source terms S (additional inflows of water or mass) and f_R (reaction terms or 'processes') to the above equation. As many source terms as required may be added (Loucks and van Beek, 2005). This equation is closely related to the incompressible Navier-Stokes equations, which consist of equations that ensure conservation of mass and conservation of momentum.

The advection-diffusion reaction equation can only rarely be solved analytically. For most real-world computational fluid dynamics problems, this equation and other governing partial differential equations are too complex to be solved analytically. Therefore, the problem must be solved numerically, most frequently using a finite volume method. With finite volume methods, the model domain is divided up into control volumes, with the value at the center of each control volume taken to be representative of all values within the control volume. Integrating the original PDE over the control volume casts the equation into a form that ensures conservation, and generates a system of sparse linear equations that can be solved using a standard linear solver method, a process termed discretization (Norris, 2000). Problems must be discretized in both space and time.

Computational fluid dynamics models can be either one-dimensional (1D), two-dimensional (2D), or three-dimensional (3D). For a mesh in a Cartesian coordinate system, the 2D and 3D discretizations are composed of two or three (respectively) orthogonal 1D discretizations along each axis of the model domain (Norris, 2000). While 3D discretization allows for the greatest model complexity, and thus allows for the greatest accuracy in representing real-world problems (given sufficient constraints), it is also the most computationally intensive, and certain systems may be simplified to 1D or 2D with minimal loss of accuracy.

According to MacWilliams et al. (2006), 1D models are able to represent tides, water levels, and depth-averaged temperature, and to partially represent tidal trapping, sediment routing, erosion and deposition, and passive and active particle trapping. Moving to a 2D model adds the ability to represent mixing in open water embayment and wind waves, to fully represent tidal trapping and sediment routing, and to partially represent mixing at junctions and wind-driven circulation. All of the above-mentioned physical processes can be fully represented by a 3D model, along with temperature stratification and gravitational circulation / salinity intrusion.

While SRH-2D is typically applied to rivers and has not previously been applied to the Sacramento-San Joaquin River Delta (Delta), according to Martyr-Koller et al. (2017), hydrodynamics and scalar transport in the Delta can be well described by a 2D model. Several 2D models have been applied to the Delta, including RMA2 (King, 1990) and 2D implementations of the 3D models Delft 3D Flexible Mesh (Achete et al., 2015) and Trim3D (Monsen et al., 2007).

Key factors in obtaining an accurate, stable solution for finite volume computational fluid dynamics problems include the choice of discretization scheme and the mesh geometry. Decreasing the size of the mesh and the size of the time step increases the accuracy of the solution, but also increases the computational time. Additionally, some schemes (primarily the so-called explicit time discretization schemes) produce stable schemas only with sufficiently low values of the Courant number, C, defined in one dimension as:

$$C = \frac{u\Delta t}{\Delta x} \tag{5}$$

where u is the characteristic velocity of the system, Δt is the time step, and Δx is the grid spacing. Thus, for these schemes, as velocity increases or mesh size decreases, the time step must be decreased to ensure stability. The maximum value of the Courant number that ensures stability is referred to as the Courant-Friedrichs-Lewy (CFL) condition.

Discretization schemes can be classified by their order (first, second, or third), frame of reference (Eulerian, Lagrangian, or semi-Lagrangian) or explicitness (fully explicit, fully implicit, or semi-implicit).

The order of a discretization scheme is defined by how many terms of a Taylor Series expansion are included. First-order schemes are the simplest but tend to be overly diffusive, leading to solutions that are both quantitative and qualitatively incorrect. Higher order schemes are more accurate, but their solutions can contain nonphysical oscillations. Many higher order schemes apply damping to prevent oscillations.

In an Eulerian specification of the field, the governing equations are discretized in time using a fixed frame of reference. They are generally the easiest methods to understand and code. With Lagrangian specification of the field, the equations are written along a moving frame of reference, following the motions of an individual parcel of fluid (Giraldo, year unknown).

Explicit schemes are generally less computationally intensive to solve than implicit schemes but must have a Courant number less than 1 to ensure stability. Semi-implicit schemes attempt to balance stability and computational speed by solving some terms implicitly and others explicitly.

2.2.2 Analytical Solutions to the Scalar Transport Equation

As stated in section 2.2.1, most real-world transport problems are too complex to be solved analytically. However, analytical solutions are available for some simple problems. These simple problems can be used for verification and error analysis of numerical models. Most analytical solutions are for 1D problems and/or problems on infinite or semi-infinite domains (Pérez Guerrero et al., 2009), which are unsuitable for the numerical verification of SRH2D, but solutions do exist for 2D/3D problems on a finite domain.

Pérez Guerrero et al. (2009) solved the scalar transport equation using a change-of-variable and integral transform technique. Their solution is valid for three-dimensional linear problems in a finite domain with decay and source terms, with any combination of type 1 (Dirichlet), type 2 (Neumann), or type 3 (Cauchy) boundary conditions. Notation in the section below is changed from the original source for consistency with equation (4).

For advection-diffusion in a transient regime governed by the equation

$$R\frac{\partial C}{\partial t} = D_x\frac{\partial^2 C}{\partial x^2} - v_x\frac{\partial C}{\partial x} + D_y\frac{\partial^2 C}{\partial y^2} - v_y\frac{\partial C}{\partial y} + D_z\frac{\partial^2 C}{\partial z^2} + v_z\frac{\partial C}{\partial z} - \lambda Rt \qquad (6)$$

the full solution is:

$$C(x,y,z,t) = \theta(x,y,z,t)\exp\left\{\frac{v_x x}{2D_x} + \frac{v_y y}{2D_y} + \frac{v_z z}{2D_z} - t\left[\left(\frac{{v_x}^2}{4D_x R} + \lambda\right) + \frac{{v_y}^2}{4D_y R} + \frac{{v_z}^2}{4D_z R}\right]\right\} + F(x,y,z;t)$$

where

$$\theta(x,y,z,t) = \sum_{i=1}^{\infty} \widetilde{\psi}_i(x,y,z)\exp\left(-\frac{{\mu_i}^2}{R}t\right)\left[\bar{f}_i + \frac{1}{R}\int_0^t \overline{G_i}(\tau)\exp\left(\frac{{\mu_i}^2}{R}\tau\right)d\tau\right],$$

$$\bar{f}_i = \int_V \widetilde{\psi}_i(x,y,z)\frac{\rho(x,y,z) - F(x,y,z;0)}{\exp(p_1 x + p_2 y + p_3 z)}d\bar{v}$$

and v_x, v_y, and v_z are the advection components, D_x, D_y, and D_z are the diffusion constants, λ is the generic decay constant, $F(x,y,z;t)$ is any equation satisfying the original boundary conditions of $T(x,y,z,t)$, $\widetilde{\psi}_i$ is a normalized eigenfunction, μ_i is an eigenvalue, R is a coefficient, $\overline{G}$ is the integral transform of the source term G, τ is an auxiliary variable, ϱ is the initial condition, V is the a generic finite volume, and p_1, p_2, and p_3 are constants for the algebraic substitution. The derivation of this solution is included in Pérez Guerrero et al. (2009).

A comparison between equation (4) and equation (6) shows that, for a salinity transport problem, the diffusion constant λ is 0 and the coefficient R is 1, thus the solution simplifies somewhat to:

$$C(x,y,z,t) = \theta(x,y,z,t)\exp\left\{\frac{v_x x}{2D_x} + \frac{v_y y}{2D_y} + \frac{v_z z}{2D_z} - t\left[\frac{{v_x}^2}{4D_y} + \frac{{v_y}^2}{4D_y} + \frac{{v_z}^2}{4D_z}\right]\right\} + F(x,y,z;t)$$

While this solution is theoretically applicable to problems with time-dependent boundary conditions, the need to define the filter function F in both time and space can make its application to such problems in a finite domain very difficult (Chen and Liu, 2011). While the generality of this solution is useful for its application to a large variety of problems, it can be difficult to evaluate.

Zoppou and Knight (1999) determined analytical solutions that are easy to evaluate for four different 2D scalar transport problems. The problem they considered is for a line source of unit strength and for corner flow.

They solved the scalar transport for four different scenarios: an instantaneous or steady release of contaminant, and with or without an impermeable boundary. These solutions are simple enough that they can be easily evaluated in a spreadsheet software such as Microsoft Excel.
The solution for the concentration following an instantaneous unit line release is:

$$C(x,y,t) = \frac{1}{4\pi D_0 u_0 t\sqrt{xyx_0y_0}}\left(\frac{xy_0}{x_0y}\right)^{\frac{1}{(2u_0D_0)}}\exp\left(\frac{-\rho^2 - 2(1 + D_0^2u_0^2)t^2}{4D_0t}\right) \quad (7)$$

where

$$\rho = \frac{1}{u_0}\sqrt{\ln^2\left(\frac{x}{x_0}\right) + \ln^2\left(\frac{y}{y_0}\right)} \quad (8)$$

and D_0 is the constant diffusion coefficient, $u_0$is the constant velocity, and x_0 and y_0 are the locations of the of the initial release of contaminant in the x and y directions, respectively,

This solution was implemented in Microsoft Excel, using values of $D_0 = 2$, $u_0 = 1$, $x_0 = 5$, and $y_0 = 5$. For simplicity, a 20-by-20 grid was used, but the solution is applicable to any non-zero values of x and y, so it could also be applied to an arbitrary unstructured grid. An example of the solution from this implementation at $t = 0.05$ is shown in Figure 3.

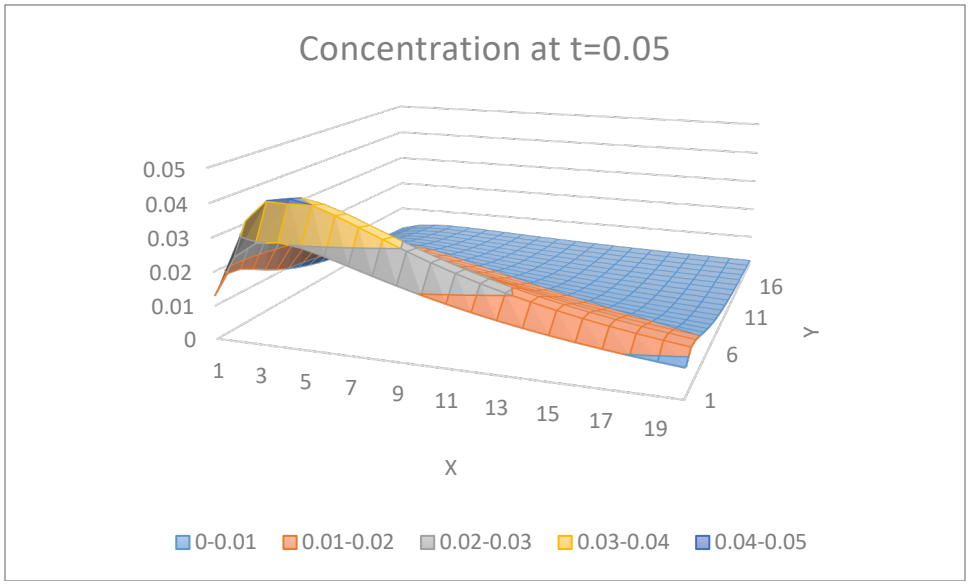


Figure 3. Contaminant concentration at $t = 0.05$ using the solution of Zoppou and Knight (1999) for an instantaneous unit line release located at $x_0 = 5$, $y_0 = 5$, using values of $D_0 = 2$ and $u_0 = 1$.

2.2.3 Summary of SRH2D

SRH2D is a 2D unstructured hybrid mesh numerical model that was developed to simulate open channel flows (Lai, 2010), following the 3D work of Lai et al. (2003).

The governing equations are depth-averaged Navier-Stokes equations, which are solved for steady or unsteady incompressible turbulent flows. Depth averaging may be performed because most open channels have shallow depths and negligible vertical motion (Lai, 2010).

The equations are discretized using the segmented finite volume approach. An element-centered scheme is used, meaning that all dependent variables are located at the centroid of the element instead of at the element vertices. An arbitrarily shaped element can be divided into triangles by connecting the center point of the element to all vertices. Thus, the equations may be discretized over triangular faces without loss of generality, as the solutions for a triangular face may be summed across all triangles comprising an element (Lai, 2000).

The cell face diffusive flux at an element face is calculated by defining a local nonorthogonal coordinate system, allowing the diffusive flux to be split into a normal term and a cross term. The cross term may be transformed to a line integral along the perimeter of the face using Green's theorem. The resulting cross term is second-order in accuracy, while the normal term is second order only for select mesh geometries (Lai, 2000).

Cell face values are calculated using the procedure proposed by Rhie and Chow and Peric et al., in which the face normal velocity is obtained by averaging the momentum equation from element centers to element faces (Lai, 2000). SRH-2D uses the nRT1 method to obtain nodal velocities from normal velocities. In this method, nodal velocities are calculated using RT0 basis functions for node *p* using the two adjacent edges. These nodal velocities differ when calculated in different cells for the same node and thus are considered "local" nodal velocities (Wang et al., 2011; Walters et al., 2007).

The pressure correction equation is derived from the mass conservation equation using a predictor-corrector algorithm called PISO. PISO can use any number of corrector steps, but two is usually sufficient (Lai, 2000).

Four boundary conditions are encountered: inlet, outlet, no-slip wall, and symmetric boundaries. Various extrapolations are needed to solve the momentum equations at the boundaries. At inlet and no-slip wall boundaries, the Cartesian velocity components are specified at boundary element face centers, and the pressure is extrapolated from the interior. At an outlet, pressure is specified at the element face center and Cartesian velocity components are extrapolated from the interior. At symmetric boundaries, the velocity component normal to the boundary is set to zero while tangential components and pressure are extrapolated from the interior (Lai, 2000).

All governing equations are solved sequentially. For a typical steady-state simulation, momentum equations are solved first assuming a known pressure field, then the predicted velocity field is used to calculate the element face normal velocity. This calculated velocity usually does not satisfy mass conservation, so the corrector steps of the PISO algorithm are applied by solving pressure correction equations. This solution process iterates until a preset convergence criterion are reached. The systems of linear equations generated by this method are solved using the preconditioned conjugate gradient method for unstructured meshes, and the strongly implicit procedure (SIP) for structured or block structured meshes (Lai, 2000).

While the solver is approximately second order using formal Taylor series analysis, the actual accuracy will be less than second order due to the use of damping in the convective flux and the reduced accuracy of the normal diffusion on nonregular mesh. For an example nonregular hexahedral mesh, the order of accuracy was estimated to be less than but close to 2.0 (1.79-1.95).

2.2.4 Comparisons of transport schemes

Gross et al. (1999) evaluated the performance of various scalar advection schemes in complex, energetic estuarine tidal flows in which advection often dominates the evolution of transported scalars. For each scheme, they used relatively simple test cases with specified steady velocity fields and uniform bathymetry, then evaluates them with a complex test case of the tidal flow of the South San Francisco Bay and compares computed with measured salinity fields.

The schemes evaluated were leapfrog-central, QUICKEST (Leonard, 1979), first-order upwind, LWlim, MPDATA (Smolarkiewicz, 1984), and ELM. The leapfrog-central scheme has no dissipation but only dispersive error, resulting in strong oscillations. The QUICKEST scheme is a variation on the popular QUICK scheme. Unlike QUICK, QUICKEST is stable for pure advection when used with explicit Euler time advancement, for $|c| \leq 1$. LWlim is a flux-limiting scheme that uses Roe's superbee limited and a second-order Lax-Wendroff scheme. It is total-variation diminishing (TVD)

and conservative, so it will not create oscillations and is guaranteed to be stable. MPDATA ensures sign preservation of scalar values and has low numerical diffusion relative to first-order upwind differencing. The original version may allow oscillations, but a later variation does not. ELM is a nonconservative scheme. Scalar fluxes are never calculated – instead, concentrations are calculated by tracing Lagrangian trajectories to determine where the water parcel at the center of a cell originated at the previous time step. This scheme is stable and oscillation-free (Gross et al., 1999).

Velocities for the advection calculations in the complex test case were obtained from the TRIM-2D model. The governing equations for hydrodynamics are the depth-averaged shallow water equations, including the continuity equation, x- and y- momentum equations, and a depth averaged conservative tracer transport equation (Gross et al., 1999).

When applied to the South San Francisco Bay Model, the Leapfrog-central method was not stable. The authors used an Asselin filter to try to stabilize it but it was inadequate. MPDATA was stable but not conservative; 40% of the initial salt mass was destroyed numerically (Gross et al., 1999).

The QUICKEST, first-order upwind, and LWlim schemes gave similar results, though QUICKEST had some oscillations. The ELM scheme did not match the field data closely, possibly due to lack of conservation. The authors concluded that stability and conservation appear to be more important than Taylor-series accuracy for the modeled application and grid size of 200 meters (Gross et al., 1999).

Another common algorithm, which was not analyzed in Gross et al.'s 1999 paper, is the SIMPLEC algorithm. SIMPLEC (Semi-Implicit Method for Pressure Linked Equations-Corrected) is a modified form of the SIMPLE algorithm and was developed by van Doormall and Raithby (1984). SIMPLEC seeks to mitigate the effects of dropping velocity neighbor correction terms, by retaining approximate versions of these terms. SIMPLEC is found to have approximately the same cost per iteration as SIMPLE but converge 20-30% faster for many problems.

2.2.5 Previous Modeling of the San Francisco Bay-Delta

The hydrologic system comprised of the San Francisco Bay and Sacramento-San Joaquin River Delta (Delta) is referred to as the San Francisco Bay-Delta. The Delta is the largest estuary on the west coast of North and South America, covering more than 1300 square miles (Isenberg et al., 2008). The Delta is connected to the San Francisco Bay by the Carquinez Strait.

Saline marine flows enter the San Francisco Bay in the west through the San Francisco channel. These flows mix with fresh water from the Delta in the east, creating a salinity gradient from west to east. The stratification of salinity varies on hourly to seasonal time-scales and is influenced by the volume of freshwater outflows and changes in the strength of tidal mixing. Salinity distribution is also affected by Pacific Ocean salinity and higher conductivity waters from the San Joaquin River and from agricultural run-off. Short-term processes such as fronts and intermittent, short-duration surface flows can also impact surface conductivity (Martyr-Kroller, et al., 2017).

The San Francisco Bay is comprised of four smaller bays, with different salinity levels. In order of increasing salinity, they are the Suisun Bay, the San Pablo Bay west of Carquinez Strait, the South Bay, and the central Bay connected to the ocean at the Golden Gate (Chao et al., 2017).

Modeling of the San Francisco Bay-Delta system began in the late 1970s with the 1D Fischer-Delta Model, with multidimensional modeling beginning in the mid-1990s (MacWilliams et al., 2006). A subset of multidimensional models that have been applied to the San Francisco Bay-Delta are described here.

2.2.5.1 UnTRIM 2D/3D

TRIM models are a family of semi-implicit finite difference schemes that have been formulated so that the gravity wave terms, transport terms, and vertical terms are treated implicitly due to their effects on stability, while the remaining terms are treated explicitly. This approach improves computational efficiency while maintaining stability. Computations are carried out over a uniform finite-difference mesh without invoking coordinate transformations. UnTRIM is a model which preserves the basic numerical properties and modeling philosophy of TRIM, but uses an unstructured orthogonal grid (Cheng and Casulli, 2002). Unstructured grids are often desirable for estuarine systems, because of the need for a large model domain, with high resolution critical near-shore but coarse resolution acceptable offshore (Shen et al., 2006).

UnTRIM uses an Eulerian-Lagrangian transport scheme for the convective terms, which does not require a CFL condition for stability (Shen et al., 2006). The boundary conditions at the bottom and free surface are considered almost flat horizontal, allowing for simplification of the tangential stress boundary conditions for the momentum equations. This simplification is valid for more environmental problems, in which the vertical scale is much smaller than the horizontal scale (Casulli and Zanolli, 2002).

Unstructured grids in UnTRIM must be orthogonal, meaning that a line segment joining the centers of any two adjacent polygons intersects the boundary between those polygons; for example, a set of Delaunay triangles with only acute angles (Shen et al., 2006).

One limitation of UnTRIM is that it has not yet been directly coupled with sediment transport, water quality, or ecology models (Achete et al., 2015).

UnTRIM was applied to the San Francisco Bay by Cheng and Casulli (2002). MacWilliams and Cheng (2006) added grid refinement around San Pablo Bay to evaluate the effects of a proposed Aquatic Transfer Facility, and performed calibration and validation against two independent observed datasets. They found that their model accurately predicted tidal range and tidal propagation from the Pacific Ocean through Suisun Bay.

2.2.5.2 Delft3D-FM

Delft 3D Flexible Mesh (Delft3D-FM) is a semi-implicit unstructured grid finite volume model. It allows for straightforward coupling of its hydrodynamic modules with a water quality model, Delft-WAQ. Coupling occurs off-line for faster calibration and sensitivity analysis (Achete et al., 2015).

The governing equations for Delft3D-FM are the incompressible 3D Navier-Stokes equations. The transport equation is simplified by ignoring density variations. Spatial discretization is performed in a staggered manner, with velocity normal components defined at the cell edges and water levels at the cell centers. In the horizontal direction the discretization is unstructured but must be orthogonal; in the vertical direction an equidistant mesh is applied that is either fixed in space or moving with the local water column height (Martyr-Kroller et al., 2017).

Horizontal spatial discretization is performed by fist discretized advection and diffusion operators at cell centers, then interpolating them back to faces and projecting them to the face-normal direction. A higher-order limited upwind scheme is used for the cell-centered discretization of advection (Martyr-Kroller et al., 2017).

Vertical advection of momentum and turbulence properties are regarded to be less important than other terms and are discretized using a first-order upwind scheme. A higher-order scheme is used for vertical advection of transported matter, such as salt, temperature, and sediment. Temporal discretization is performed using a predictor-corrector time-step method.

Because the horizontal advective terms are discretized explicitly, a CFL condition arises. However, because of the implicit treatment of other terms, the CFL condition is based only on the horizontal advection velocity, and not on the free-surface wave propagation speed or the vertical advection velocity.

Delft-3D-FM was used to study sediment dynamics in the San Francisco Bay-Delta in 2D by Achete et al. (2015). Their model was able to reproduce the general trends in suspended sediment concentration, but in some places it was limited by its inability, as a 2D model, to represent the vertical stratification of salinity.

The Delft3D-FM was also the foundation for the Computational Assessments of Scenarios of Change for the Delta Ecosystem (CASCaDE) II study of the San Francisco Bay-Delta estuary-watershed system led by the USGS (USGS, 2015). CASCaDE II uses a linked model approach, applying a hydrodynamic model to separately drive associated sediment, water-quality, contaminants, and ecology models in a loosely coupled format (Martyr-Koller et al., 2017).

The CASCaDE II study included modeling the spatiotemporal patterns of salinity. Modeled salinity generally matched observed salinity at the tidal, seasonal, and annual scales, though there were differences in timing and magnitude of some variations. Modeled stratification matched the general pattern of measured stratification, but the modeled stratification ranges were smaller than measured ranges (Martyr-Kroller, 2017).

2.2.5.3 SCHISM

Semi-implicit Cross-scale Hydroscience Integrated System Model (SCHISM) is an open-source, semi-implicit 3D (2D optional) unstructured grid hydrodynamic model. SCHISM, formerly known as SELFE, is the working model of the California Department of Water Resources (DWR) (Ateljevich et al., 2014).

SCHISM is based on ELCIRC but its discretization and solution scheme has been modified to improve the depiction of bathymetry and salinity plume transport, and it has been parallelized for efficient computation. SCHISM also shares aspects of the UnTRIM family of models (Ateljevich et al., 2014).

The governing equations are the Navier-Stokes equations and the transport equations for salt and heat. Flow is assumed to be Reynolds-averaged. There are three algorithm options for constituent transport: first-order upwind, TVD upwind, and the Eulerian-Lagrangian Method (ELM). The first-

order upwind is faster but diffuses the vertical salinity structure, while the TVD upwind method is slower but preserves sharp gradient. These two schemes may be mixed adaptively for different locations and depths. ELM combines particle-like backtracking along the velocity field with interpolation, but is rarely used because it is not mass conservative (Ateljevich et al., 2014).

The horizontal mesh is unstructured. The mesh is limited to triangles, but those triangles are not required to be orthogonal. Vertical meshing allows for a combination of fixed depth or terrain-following layers (Ateljevich et al., 2014).

This model was applied to the San Francisco Bay-Delta by Ateljevich et al. (2014). Their model uses bathymetry from 10m and 2m digital elevation models (DEMs), includes all major gate and hydraulic structures in the Bay-Delta system, a horizontal mesh composed of 144,000 triangles that range in width from approximately 1 km in the ocean to less than 5 meters near Middle River, and a vertical mesh consisting of 23 terrain-following layers. This model was calibrated for 2009-2010. Salinity results generally followed seasonal trends well and has errors comparable to other models. The sensitivity to sustained periods of low outflows was found to be problematic. The model is able to pick up a large stratification at Richmond, but under-predicts the largest events at Benicia (Ateljevich et al., 2014).

Chao et al. (2017) also applied SCHISM to the San Francisco Bay-Delta, using the bathymetry and mesh produced by Ateljevich et al., (2014) and adding a longer period of simulation (2005-2016), coupling to a coastal Regional Ocean Modeling System (ROMS), and connection to coastal processes. This model represented the patterns of salinity variations relatively well, but like the Ateljevich et al. (2014) model, tended to underestimate salinity.

2.2.5.4 SUNTANS

SUNTANS (Stanford Unstructured Nonhydrostatic Terrain-following Adaptive Navier-Stokes Solver) is a parallel nonhydrostatic three-dimensional unstructured grid coastal ocean hydrodynamic model that uses a finite-volume formulation to solve the hydrodynamics and scalar transport simulations. Chua and Fringer (2011) used a hydrostatic implementation of SUNTANS to model salinity in North San Francisco Bay.

The governing equations are the three-dimensional, Reynolds-averaged primitive equations. The primitive equations of the ocean consist of a continuity equation, a thermal energy equation, and a form of the Navier-Stokes equation that describes hydrodynamical flow on the surface of a sphere using the Boussinesq approximation and hydrostatic approximation (Lehner, 2017).
These equations are solved using the theta-method (Casulli, 1999) to solve implicitly for free-surface height, vertical diffusion of momentum, and vertical scalar advection and diffusion, and the second-order Adams-Bashforth method for all other terms. The Eulerian-Lagrangian method is used for advection of momentum (Chua and Fringer, 2011). The momentum scheme is conservative, but introduces a CFL condition as momentum is calculated explicitly (Friger et al., 2006).

The SUNTANS model of Chua and Fringer (2006) was found to reproduce the variability in the observed currents relatively well at both Richmond and Oakland. There was also good agreement between observed and modeled salinity amplitude and phase at both of the locations considered, Benicia and Point San Pablo. The model also reproduced the observed stratification at Point San Pablo, except during one time period where the model unpredicted the stratification as a

consequence of overpredicting the minimum surface salinity. At Benicia, the modeled stratification is relatively insensitivity to the spring-neap variability, while the observed stratification is more sensitive to this parameter.

The sensitivity of the model to grid resolution was also tested. This sensitivity analysis showed that model convergence is highly sensitive to the choice of advection scheme and the turbulence model. Using the TVD scheme for salt transport, rather than first order upwinding, and including a turbulence model achieved the best convergence rate in space. Without the turbulence model, the error is about one order of magnitude higher, due to the lack of feedback between vertical mixing and stratification. The error with first-order upwinding is about twice as high, and does not decrease with mesh refinement.

2.2.5.5 TELEMAC-MASCARET

TELEMAC-MASCARET is an integrated open-source hydrodynamic simulation system that includes both a 2D module (TELEMAC -2D) and 3D module (TELEMAC-3D) (Galland, et al., 1991). Both TELEMAC-2D and TELEMAC-3D use a horizontally unstructured triangular element grid; TELEMAC-3D adds a series of model planes between the bed and surface planes.

The Navier-Stokes equations are solved base on the Operator-Splitting method, where the hyperbolic and parabolic parts of the Navier-Stokes equations are treated separately. The Method of Characteristics and Streamline Upwind Petrov-Galerkin method are used for advection to ensure mass conservation and an oscillation-free solution without excessive mesh refinement. The propagation, diffusion, and source terms are solved using the finite element method with implicit time discretization and solved by an iterative conjugate gradient method (Fernandes et al., 2001).

TELEMAC-2D was applied to the Sacramento-San Joaquin Delta by Wu et al. (2009), for the purpose of generating synthetic drifter data for incorporation into their 1D model. Because the TELEMAC modeling was not the focus of their research, it was not described in detail. However, TELEMAC has been applied to numerous other estuary systems, including the Patos Lagoon in Brazil (Fernandes et al., 2001), Scheldt Estuary in Belgium, France, and the Netherlands (Smolders et al., 2014), and the Irish Sea (Jones and Davies, 2006).

3. Methods and Results

A proof-of-concept Preconditioned Conjugate Gradient solver was implemented in C++ using CUDA and the CUBLAS library. This solver used ILU(0) preconditioning. At that time, a need was identified for Unified Memory to allow CUDA to access memory more efficiently.

CUDA Fortran compilers were obtained and tested. Early stage implementation began of CUDA Fortran in the SRH-2D solver, using a subset of functions within the PDE solvers. Early results suggested it would be possible to obtain significant speedup without requiring significant modifications to SRH-2D's modeling framework. However, upon further analysis, the results were found to be theoretically unstable. At least one new approach was attempted to try to make the results more stable, but significant progress beyond this point was not achieved.

Effort began to implement a SRH-2D salinity module, but significant progress does not appear to have been achieved.

4. References

Achete, F.M., van der Wegen, M., Roelvink, D., and B. Jaffe, 2015. A 2-D process-based model for suspended sediment dynamics: a first step towards ecological modeling. *Hydrology and Earth System Sciences* 19, 2837-2857.

Amestoy, P.R., Davis, T.A., and I.S. Duff, 1996. An approximate minimum degree ordering algorithm, SIAM Journal of Matrix Analysis and Applications 17, 896-905.

Amestoy, P.R., Davis, T.A., and I.S. Duff, 2004. Algorithm 837:AMD, an approximate minimum degree ordering algorithm, ACM Transactions on Mathematical Software 30, 381-388.

Amouzgar, R., Liang, Q., Clarke, P.J., Yasuda, T., and H. Mase, 2016. Computationally Efficient Tsunami Modeling on Graphics Processing Units (GPUs). International Journal of Offshore and Polar Engineering 26(2), 154-160.

Arslan, T., 2016. A benchmark test for OpenFOAM using GPU cards: Flow past a centrifugal pump. NTNU HPC Wiki. Available online at https://www.hpc.ntnu.no/display/hpc/A+benchmark+test+for+OpenFOAM+using+GPU+cards+%3A+Flow+past+a+centrifugal+pump. Accessed 12/31/18.

Ateljevich, E., Nam, K., Zhang, Y., Wang, R.F., and Q. Shu, 2014. Bay-delta SELFE calibration overview. In: Methodology for flow and salinity estimates in the Sacramento-San Joaquin Delta and Suisun Marsh, 35th annual progress report. Department of Water Resources.

Benzi, M., Cullum, J.K., and M. Tuma, 2000. Robust approximate inverse preconditioning for the conjugate gradient method. SIAM Journal of Scientific Computing 22, 1318-1332.

Benzi, M., Meyer, C.D., and M. Tuma, 1996. A sparse approximate inverse preconditioner for the conjugate gradient method. SIAM Journal of Scientific Computing 17, 1135-1149.

Benzi, M., and M. Tuma, 1998. A sparse approximate inverse preconditioner for the conjugate gradient method. SIAM Journal of Scientific Computing 19, 968-994.

Bertaccini, D., and S. Fillipone, 2016. Sparse approximate inverse preconditioners on high performance GPU platforms. *Computers and Mathematics with Applications* 71, 693-711.

Bolz, J., Farmer, I., Grinspun, E., and P. Schröder, 2003. Sparse Matrix Solvers on the GPU: Conjugate Gradient and Multigrid. *ACM Transactions on Graphics (TOG)* 22(3), 917-924.

Cai, X.-C., and M. Sarkis, 1999. A restricted additive Schwarz preconditioner for general sparse linear systems, SIAM Journal of Scientific Computing 21, 792-797.

Cai, X.-C., Farhat, C., and Sarkis, M., 1998. A Minimum Overlap Restricted Additive Schwarz

Preconditioner and Applications in 3D Flow Simulations. Contemporary Mathematics 218, 479-485.

Casulli, V., 1999. A Semi-Implicit Finite-Difference Method for Non-Hydrostatic, Free-Surface Flows. International Journal for Numerical Methods in Fluids 30, 425-440.

Casulli, V. and P. Zanolli, 2002. Semi-Implicit Numerical Modeling of Nonhydrostatic Free-Surface Flows for Environmental Problems. *Mathematical and Computer Modeling* 36, 1131-1149.

Chang, L.W., and Hwu, W.-m.W., 2014. A Guide for Implementing Tridiagonal Solvers on GPUs. In Kindratenko, V. (Ed.), *Numerical Computation with GPUs*, Springer International Publishing Switzerland.

Chao, Y., Farrara, J.D., Zhang, H., Zhang, Y.J., Ateljevich, E., Chai, F., Davis, C.O., Dugdale, R., and F. Wilkerson, 2017. Development, implementation, and validation of a modeling system for the San Francisco Bay and Estuary. *Estuarine, Coastal and Shelf Science* 194, 40-56.

Chen, J.-S. and C.-W. Liu, 2011. Generalized analytical solution for advection-dispersion equation in finite spatial domain with arbitrary time-dependent inlet boundary condition.

Chen, Z., Liu, H., and B. Yang, 2016. Parallel Triangular Solvers on GPU. *arXiv preprint arXiv: 1606.0054*.

Cheng, R.T., and V. Casulli, 2002. Evaluation of the UnTRIM Model for 3-D Tidal Circulation. Proceedings of the 7th International Conference on Estuarine and Coastal Modeling, St. Petersburg, FL, 628-642.

Chua, V.P. and O.B. Fringer, 2011. Sensitivity analysis of three-dimensional salinity simulations in North San Francisco Bay using the unstructured-grid SUNTANS model. *Ocean Modelling* 39, 332-350.

Fernandes, E.H., Dyer, K.R., and L.F.H. Niencheski, 2001. Calibration and Validation of the TELEMAC-2D Model to the Patos Lagoon (Brazil). *Journal of Coastal Research*, 470-488.

Fillipone, S., and A. Buttari, 2018. PSBLAS 3.6.0 User's guide: A reference guide for the Parallel Sparse BLAS library.

Fillipone, S., Cardellini, V., Barbieri, D., and A. Fanfarillo, 2017. Sparse Matrix-Vector Multiplication on GPGPUs. ACM Transactions on Mathematical Software 43(3), Article 30.

Galland, J.-C., Goutal, N., and J.-M. Hervouet, 1991. TELEMAC: A new numerical model for solving shallow water equations. *Advances in Water Resources* 14(3), 138-148.

Geveler, M., Ribbrock, D., Goddeke, D., Zajac, P., and S. Turek, 2011. Towards a complete FEM-based simulation toolkit on GPUs: Unstructured Grid Finite Element Geometric Multigrid solvers with strong smoothers based on Sparse Approximate Inverses.

Gibbs Jr., N.E., Poole, W.G., and P.K. Stockmeyer, 1976a. A comparison of several bandwith and profile reduction algorithms. ACM Transactions on Mathematical Software 2, 322-330.

Gibbs Jr., N.E., Poole, W.G., and P.K. Stockmeyer, 1976b. An algorithm for reducing the bandwidth and profile of a sparse matrix. SIAM Journal of Numerical Analysis 18, 235-251.

Giraldo, F.X., year unknown. Time-Integrators. Available online at http://faculty.nps.edu/fxgirald/projects/nseam/nps/new_section4.pdf. Accessed 10/2/18.

Govindaraju, N.K., Larsen, S., Gray, J., and D. Manocha, 2006. A Memory Model for Scientific Algorithms on Graphics Processors. Microsoft Technical Report MSR TR 2006 108, 10 pp.

Gross, E.S., Koseff, J.R., and S.G. Monismith, 1999. Evaluation of Advective Shemes for Estuarine Salinity Simulations. *Journal of Hydraulic Engineering* 125(1), 32-46.

Ha, S., Park, J., and D. You, 2018. A GPU-accelerated semi-implicit fractional-step method for numerical solutions of incompressible Navier-Stokes equations. Journal of Computational Physics 352, 246-264.

He, B., Govindaraju, N.K., Luo, Q., and B. Smith, 2007. Efficient Gather and Scatter Operations on Graphics Processors. Proceedings of the 2007 ACM/IEEE conference on Supercomputing, 10 pp.

Helfenstein, R., and J. Koko, 2012. Parallel preconditioned conjugate gradient algorithm on GPU. Journal of Computational and Applied Mathematics 236, 3584-3590.

Hestenes, M.R., and E. Stiefl, 1952, Methods of conjugate gradients for solving linear systems. Journal of Research for the National Bureau of Standards 49, 409-436.

Heuveline, V., Lukarski, D., and J.-P. Weiss, 2011. Enhanced Parallel ILU(p)-based Preconditioners for Multi-core CPUs and GPUs – The Power(q)-pattern Method. Preprint series of the Engineering Mathematics and Computing Lab (EMCL), No. 2011-08.

Isenberg, P., Florian, M., Frank, R., McKernan, T., McPeak, S., Reilly, W., and R. Seed, 2008. Blue Ribbon Task Force Delta Vision: Our Vision for the California Delta. Sacramento, California: State of California Resources Agency.

Jasak, H., 1996. Error Analysis and Estimation for the Finite Volume Method with Applications to Fluid Flows. PhD Dissertation, Imperial College, London.

Jones, J.E. and A.M. Davies, 2006. Application of a finite element model (TELEMAC) to computing the wind induced response of the Irish Sea. *Continental Shelf Research* 26(12-13), 1519-1541.

Kao, N.S.-C. and Sheu, T.W.-H., 2018. Development of a finite element flow solver for solving three-dimensional incompressible Navier-Stokes solutions on multiple GPU cards. Computers and Fluids 167, 285-291.

King, I., 1997. RMA2 – A Two Dimensional Finite Element Model for Flow in Estuaries and Streams, v. 6.4. Davis, CA: Department of Civil Engineering, University of California, Davis.

Kreutzer, M., Hager, G., Wellein, G., Fehske, H., Basermann, A., and A. Bishop, 2012. Sparse Matrix-vector Multiplication on GPGPU Clusters: A New Storage Format and Scalable Implementation. Workshop on Large-Scale Parallel Processing held at the IEEE International Parallel and Distributed Processing Symposium 2012.

Lai, Y.G., 1997. An Unstructured Grid Method for a Pressure-Based Flow and Heat Transfer Solver. Numerical Heat Transfer 32(3), 267-281.

Lai, Y.G., 2000. Unstructured Grid Arbitrarily Shaped Element Method for Fluid Flow Simulation. *AIAA Journal* 38(12), 2246-2252.

Lai, Yong, 2008. SRH-2D version 2: Theory and User's Manual. Denver, CO: Bureau of Reclamation Technical Service Center, Sedimentation and River Hydraulics Group.

Lai, Y.G., 2010. Two-Dimensional Depth-Averaged Flow Modeling with an Unstructured Hybrid Mesh. Journal of Hydraulic Engineering 136(1), 12-23.

Lai, Y.G., Weber, L.J, and V.C. Patel, 2003. Nonhydrostatic Three-Dimensional Model for Hydraulic Flow Simulation. I: Formulation and Verification. *Journal of Hydraulic Engineering* 129(3), 196-205.

Lefohn, A., Kniss, J., and Owens, J., 2005. Chapter 33. Implementing Efficient Parallel Data Structures on GPUs. *In* Pharr, M. (ed.). GPU gems 2: programming techniques for high-performance graphics and general-purpose computation. NVIDIA Corporation. Available online at https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter33.html. Accessed 11/7/18.

Lehner, M., 2017. Oceanic and Atmospheric Fluid Dynamics. Bachelor Thesis, Johannes Kepler Univeresität Linz.

Leonard, B.P., 1979. A stable and accurate convective modeling procedure based on quadratic upstream interpolation. Computer Methods in Applied Mechanics and Engineering 19, 59-98.

Li, R., and Y. Saad, 2010. GPU-Accelerated Preconditioned Iterative Linear Solvers. Technical Report UMSI-2010-112, University of Minnesota Supercomputing Institute, Minneapolis, MN.

Liu, H., Chen, Z., and B. Yang, 2014. Accelerating Preconditioned Iterative Linear Solvers on GPU. *International Journal of Numerical Analysis and Modeling, Series B* 5(1-2), 136-146.

Liu, Q., Qin, Y., and G. Li, 2018. Fast Simulation of Large-Scale Floods Based on GPU Parallel Computing, Water 10, 589, 16 pp.

Liu, X.-X., Zhai, K., Liu, Z., He, K., Tan, S.X.-D., and Yu, W., 2015. Parallel Thermal Analysis of 3-D Integrated Circuits with Liquid Cooling on CPU-GPU Platforms.

Loucks, D.P. and E. van Beek, 2005. Water Resources Systems Planning and Management: An Introduction to Methods, Models, and Applications. Paris, France: United Nations Educational, Scientific, and Cultural Organization.

MacWilliams, M.L., Ateljevich, E.S., Monismith, S.G., and C. Enright, 2016. An Overview of Multi-Dimensional Models of the Sacramento-San Joaquin Delta. *San Francisco Estuary and Watershed Science* 14(4), article 2, 35 pp.

MacWilliams, M.L., and R.T. Cheng, 2006. Three-dimensional hydrodynamic modeling of San Pablo Bay on an unstructured grid. The 7th International Conference on Hydroscience and Engineering (ICHE-2006), September 2006.

Maggioni, M., Berger-Wolf, T., and J. Liang, 2013. GPU-based steady-state solution of the chemical master equations. In *Proceedings of IEEE 27th Intlernational Parallel and Distributed Processing Symposium Workshops and PhD Forum*, IPDPSW '13, 579-588.

Martyr-Krolloer, R.C., Kernkamp, H.W.J., van Dam, A., van der Wegen, M., Lucas, L.V., Knowles, N., Jaffe, B., and T.A. Fregoso, 2017. Application of an unstructured 3D finite volume numerical model to flows and salinity dynamics in the San Francisco Bay-Delta. *Estuarine, Coastal and Shelf Science* 192, 86-107.

Michels, D., 2011. Sparse-Matrix-CG-Solver in CUDA. Proceedings of CESCG 2011: The 15th Central European Seminar on Computer Graphics.

Miller, C.T., Dawson, C.N., Farthing, M.W., Hou, T.Y., Huang, J., Kees, C.E., Kelley, C.T., and H.P. Langtangen, 2013. Numerical simulation of water resources problems: Models, methods, and trends. Advances in Water Resources 51, 405-437.

Monakov, A., Lokhmotov, A., and A. Avetisyan, 2010. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *High Performance Embedded Architectures and Compilers*, volume 5952 of LNCS, Springer-Verlag, 111-125.

Müller, E., Guo, X., Scheichl, R., and S. Shi, 2014. Matrix-free GPU implementation of a preconditioned conjugate gradient solver for anisotropic elliptic PDEs. arXiv:1302.7193v1.

Naumov, M., 2011. Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU. NVIDIA Technical Report NVR-2011-001.

Naumov, M., 2018. Incomplete-LU and Cholesky Preconditioned Iterative Methods using CuSPARSE and CuBLAS. NVIA White Paper WP-06720-001-v.10.0, 24 pp.

Norris, S.E., 2000. A Parallel Navier Stokes Solver for Natural convection and Free Surface Flow. Chapter 2 Finite Volume Differencing Schemes.
Perkin, R.G. and E.L. Lewis, 1980. The Practical Salinity Scale 1978: Fitting the Data. *IEEE* Journal of Oceanic Engineering OE-5(1), 9-15.

NVIDIA, 2018a. cuSPARSE. In CUDA Toolkit Documentation. Available online at https://docs.nvidia.com/cuda/cusparse/index.html. Accessed 1/2/19.

NVIDIA, 2018b. Incomplete-LU and Cholesky Preconditioned Iterative Methods using cuSPARSE and cuBLAS. In CUDA Toolkit Documentation. Available online at https://docs.nvidia.com/cuda/incomplete-lu-cholesky/. Accessed 1/3/19.

Pérez Guerrero, J.S., Pimentel, L.C.G., Skaggs, T.H., and M.Th. van Genuchten, 2009. Analytical solution of the advection-diffusion transport equation using a change-of-variable and integral transform technique. *International Journal of Heat and Mass Transfer* 52, 3297-3304.

Phillips, E., and M. Fatica, 2016. Performance analysis of the high-performance conjugate gradient benchmark on GPUs. The International Journal of High Performance Computing Applications 30(1), 28-38.

Polizzi, E., and A.H. Sameh, 2006. A parallel hybrid banded system solver: the SPIKE algorithm. Parallel Computing 32(2), 177-194.

Robert, Y., 1982. Regular incomplete factorizations of real positive definite matrices. Linear Algebra and its Applications 48, 105-117.

Saad, Y., and M.H. Schultz, 1986. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. SIAM Journal of Scientific and Statistical Computing 7(3), 856-869.

Sameh, A.H., and D.J. Kuck, 1978. On stable parallel linear system solvers. Journal of the ACM 25(1), 81-91.

Shen, J., Wang, H., Sisson, M., and W. Gong, 2006. Storm tide simulation in the Chesapeake Bay using an unstructured grid model. *Estuarine, Coastal, and Shelf Science* 68, 1-16.

Smolarkiewicz, P., 1984. A fully multidimensional positive definite advection transport algorithm with small implicit diffusion. *Journal of Computational Physics* 54(2), 325-362.

Smolders, S., Maximova, T., Vanlede, 2014. Implementation of controlled reduced tide and flooding areas in the TELEMAC 3D model of the Scheldt Estuary. In: Bertand, O., et al. (Ed.), Proceedings of the 21st TELEMAC-MASCARET User Conference, October 2014, 111-118.

Tomczak, T., Zadarnowska, K., Koza, Z., Matyka, M., and Miroslaw, L., 2013. Acceleration of iterative Navier-Stokes solvers on graphics processing units. International Journal of Computational Fluid Dynamics 27(4:5), 201-209.

USGS, 2015. CASCaDE II Project Final Report. Menlo Park, CA: USGS, 307 pp.

Van Doormaal, J.P. and G.D. Raithby, 1984. Enhancements of the SIMPLE method for predicting incompressible fluid flows. *Numerical Heat Transfer* 7, 147-163.

van Duin, A.C.N., 1999. Scalable parallel preconditioning with the sparse approximate inverse of

approximate triangular matrices, SIAM Journal of Matrix Analyis and Applications 20, 987-1006.

van Genuchten, M.Th., Leij, F.J., Skaggs, T.H., Toride, N., Bradford, S.A., and E.M. Ponteidero, 2013. Exact analytical solutions for contaminant transport in rivers: 1. The equilibrium advection-dispersion equation. *Journal of Hydrology and Geomechanics* 61(2), 146-160.

van Oosten, J., 2011. CUDA Memory Model. 3D Game Engine Programming. Available online at https://www.3dgep.com/cuda-memory-model/. Accessed 11/29/18.

Wang, B., Zhao, G., and O.B. Fringer, 2011. Reconstruction of vector fields for semi-Lagrangian advection on unstructured, staggered grids. *Ocean Modelling* 40(1), 52-71.

Weller, H.G., Tabor, G., Jasak, H., and C. Fureby, 1998. A Tensorial Approach to Copmutational Continuum Mechanics Using Object-Oriented Techniques. Computers in Physics 12(6), 620-631.

Wu, Q., Rafiee, M., Tinka, A., and Bayen, A.M., 2009. Inverse Modeling for Open Boundary Conditions in Channel Network. Joint 48th IEEE Conference on Decision and Control and 28th Chinese Control Conference, Shanghai, P.R. China, December 2009.

Wu, W., Sanchez, A., and M. Zhang, 2011. An Implicit 2-D Shallow Water Flow Model on Unstructured Quadree Rectangular Mesh. In: Roberts, T.M., Rosatti, J.D., and Wang, P. (eds.), *Proceedings, Symposium to Honor Dr. Nicholas C. Kraus, Journal of Coastal Research*, Special Issue 59, 15-26.

Yang, B., Liu, H., Zhong, H., and C. Zhangxin, 2017. Decoupled Block-Wise ILU(k) Preconditioner on GPU. arXiv preprint arXiv:1703.01325.

Ye, F., Zhang, Y.J., Wang, H.V., Friedrichs, M.A.M., Irby, I.D., Ateljevich, E., Valle-Levinson, A., Wang, Z., Huang, H., Shen, J., and J. Du, 2018. A 3D unstructured-grid model for Chesapeake Bay: Importance of bathymetry. *Ocean Modelling* 127, 16-39.

Zhang, Y., Baptista, A.M., and E.P. Myers, 2004. A cross-scale model for 3D baroclinic circulation in estuary-plume-shelf systems: I. Formulation and skill assessment. Continental Shelf Research 24(18), 2187-2214.

Zoppou, C., and Knight, J.H., 1999. Analytical solution of a spatially variable coefficient advection-diffusion equation in up to three dimensions. *Applied Mathematical Modeling* 23, 667-685.