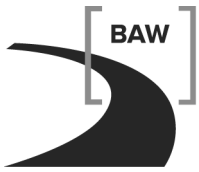

Bundesanstalt für Wasserbau
Kompetenz für die Wasserstraßen

FuE-Vorhaben

Adaptierung und Erweiterung von Casulli-Algorithmen für Parallelrechner mit Hardware-Beschleunigung und zur Anwendung von konservativen Advektionsverfahren

Abschlussbericht

A39530270001

Bundesanstalt für Wasserbau
Kompetenz für die Wasserstraßen

FuE-Vorhaben

Adaptierung und Erweiterung von Casulli-Algorithmen für Parallelrechner mit Hardware-Beschleunigung und zur Anwendung von konservativen Advektionsverfahren

Abschlussbericht

Auftraggeber:	BAW
Auftrag vom :	26. Januar 2010
Auftrags-Nr.:	BAW-Nr. A39530270001
Aufgestellt von	Abteilung W Referat W2 und W5 ab 2012
Bearbeiter	Jacek A. Jankowski

Karlsruhe, im *Januar* 2013

Bundesanstalt für Wasserbau · Kußmaustr. 17 · D-76187 Karlsruhe · ([+49] (0)721 9726 - 0

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Zusammenfassung

Das in diesem Dokument behandelte Vorhaben wurde in den Jahren 2009-13 bearbeitet. Zu dieser Zeit haben sich neue Anforderungen ergeben und mehrere Veränderungen bezüglich der Entwicklungsarbeiten an den Modellen für hochauflösende mesoskalige Modellierung in der Abteilung Wasserbau stattgefunden. Dazu noch war das die Zeit des Übergangs in den gängigen Programmier-Methoden von homogenen zu heterogenen Rechnerarchitekturen, welcher auf breiten Spektrum – von Großrechnern bis zu mobilen Geräten – weitgehend verwirklicht wurde.

Diese entstandene Unsicherheit sowohl intern im Hause, als auch angesichts der fortlaufenden Evolution von Programmier-Methoden und Entwicklungswerkzeugen hat dazu geführt, dass es einen gewissen Unterschied zwischen den 2009 geplanten Arbeiten und den im 2013 gelieferten Ergebnissen gibt. Diese Aspekte werden genauer in Hinblick auf die Motivation zu diesem Projekt im Kapitel 1 und insbesondere in Abschnitten 1.4.3 und 1.4.4 erörtert.

Das Vorhaben, für 2009 anvisiert, kann in folgenden drei Punkten kurz beschrieben werden (aus dem Antrag):

- Erarbeitung und Anwendung neuer Programmier-Paradigmen im Hochleistungsrechnen durch die Adaptierung von Casulli-Algorithmen für die neuen Rechnerarchitekturen der Parallelrechner mit Hardware-Beschleunigung. Dies sind zur Zeit überwiegend Cluster, die aus Rechen-Knoten mit Multi-Kern-CPUs und GPUs (*Graphic Processing Units*) bestehen.

- Exemplarisch soll dies für das TRIM-Verfahren erfolgen, bei dem die Struktur der Diskretisierung und bisherige BAW-Investitionen in Hardware-angepasste Programmstruktur – das Verfahren ist für parallele Vektorrechner angepasst – die Erfolgsaussichten für eine erfolgreiche Implementation nach Fach-Diskussion in internen zuständigen Gremien (VBC2/3) als maximal angesehen werden. TRIM erlaubt die Nutzung aller Ebenen der heterogenen Parallelität, welche die neuen Rechnerarchitekturen bieten. Die Erfahrungen mit den neuen Programmier-Paradigmen sollen auch auf andere Verfahren, z.B. UnTRIM, direkt übertragbar sein.

- Entwicklung neuer Advektionsverfahren zur Erfassung der lokal und räumlich begrenzt auftretenden Übergänge in der Abflussart (schießen/strömen bzw. über-/unterkritisch), die bisher teilweise außerhalb des Anwendbarkeitsbereiches der o.g. Verfahren liegen. Dabei werden die günstigen Eigenschaften der strukturierten Netze (wie in TRIM vorhanden) in Hinsicht auf die Implementation ausgenutzt.

Wegen fortlaufenden Veränderungen an der Richtung des Vorhabens und den Anforderungen an die technisch-wissenschaftliche Programmierung in der Abt. Wasserbau wurde 2013 verlangt, zwei wesentlichen Resultate dises Projekts vorzustellen. Dies sind:

1. **Ein vollständig für GPUs adaptiertes nicht-lineares, vertikal gemitteltes 2DV-Schema nach Casulli.** Es wurde gezeigt, dass eine erfolgreiche und vollständige Ad-

aptierung eines zweidimensionalen numerischen semi-impliziten Verfahrens für Strömungen mit freier Oberfläche für die Hardware-beschleunigte Ausführung auf GPUs möglich ist. Dies ist mit einem relativ geringem Aufwand geschehen und bringt gute Resultate, nämlich eine 20- oder 30-fache Geschwindigkeitssteigerung jeweils für die doppelte bzw. einfache Genauigkeit verglichen mit einem einzigen CPU-Kern. Der Schlüssel für diesen Erfolg ist einerseits die feinkörnige Parallelität im Verfahren durch geeignete Datenstrukturen für Operationen auf Datenvektoren hervorzuheben, andererseits große und zusammenhängende Anteile des Verfahrens ausschließlich auf der Grafikkarte auszuführen. In diesem Fall ist dies die gesamte Zeitschleife mit der Assemblierung von Matrizen, das Lösen des linearen Gleichungssystems für die Wasserstände eingebettet in den nicht-linearen äußeren Iterationen nach Newton für Überflutungen, sowie alle notwendigen Aktualisierungen der Variablen. Bei der in den Untersuchungen verwendeten Grafikkarte ist es möglich, abhängig von der Genauigkeit (einfach bzw. doppelt) Berechnungen bis zu ca. 4,5 bzw. 9,5 Millionen Netz-Polygone durchzuführen. Allerdings musste für diese Ergebnisse das Verfahren neu in CUDA C/C++ geschrieben werden, mit der Nutzung der bisher existierenden Implementierungen (insb. TRIM) nur als Referenz.

Die mathematischen Grundlagen des Verfahrens werden in Kapitel 2 dargestellt, seine Implementierung für GPUs in Kapitel 3 und die Ergebnisse bezüglich der Verifizierung und erzielten Geschwindigkeitsteigerungen in der Ausführung in Kapitel 4. Schlussfolgerungen befinden sich in Kapitel 6, insb. Abs. 6.2.

2. **Eine fundierte Analyse von Möglichkeiten der Adaptierung der existierenden Software für heterogene Parallelrechner.** Es wird die Software-Struktur von UnTRIM und die bisherigen HPC-Anpassungen am mathematischen Rechenkern des Verfahrens kurz dargestellt. Ausgehend von der derzeitigen Situation und den Erfahrungen aus der Hardware-Anpassung des o.g. zweidimensionalen Schemas nach Casulli wurden verschiedene, sich gegenseitig nicht ausschließende Möglichkeiten der Adaptierung von UnTRIM – vom Rechenkern und dessen Umgebung – vorgestellt. Einerseits wird es die Fortführung mit dem existierenden Code der erprobten MPI-Parallelisierung basierend auf der Datenparallelität als prinzipielle Quelle der Performance auf einem klassischen Rechner-Cluster vorhersehen, ergänzt durch neue Versuche mit OPENACC- bzw. OPENMP-Technologie, um eine Beschleunigung der einzelnen MPI-Prozesse auf Rechen-Knoten mit GPUs bzw. *multicore*-Co-Prozessoren zu erreichen. Andererseits wird eine Vorgehensweise vorgeschlagen, bei der man vom Anfang an die feinkörnige und grobkörnige Parallelität im Verfahren hervorhebt, um gezielt einen neuen Code für heterogene Parallelrechner (multi-Node, multi-GPU) mit CUDA oder OpenCL und MPI zu entwickeln. Hiermit wäre auch die Gelegenheit gegeben, ganz neue numerische Methoden im Rechenkern einzubeziehen.

   Die Diskussion der vorhandenen Möglichkeiten für die existierende Software findet man in Kap. 5 und den Schlussfolgerungen in Abs. 6.3 des Kapitels 6.

1 Einführung

1.1 Vorwort: Motivation

Die Verfügbarkeit von fein aufgelösten digitalen Gelände-Modellen (DGM) mit der horizontalen $O(1\text{m})$ und vertikalen $O(0.1\text{m})$ Auflösung, erlauben zusammen mit mit Daten über die Gelände-Beschaffenheit und genauen Luftbildern eine wesentliche Verbesserung der Genauigkeit von der numerischen Modellierung mesoskaliger Umwelt-Strömungen. Hiermit steigen auch Anforderungen an die numerischen Verfahren, bzw. zeigen sich andere als bisher bekannten Schwerpunkte bei der numerischen Modellierung.

In der letzten Dekade sind die neuen damit verbundenen Tendenzen und Aktivitäten zu beobachten, in denen die BAW bereits teilweise involviert ist. Dies spiegelt sich nicht nur in der Verwendung von hoch auflösenden Flussmodellen mit bereits existierenden Programmen unter Verwendung von parallelen Hochleistungsrechnern wider [40, 52], sondern auch in einigen praxisorientierten FuE-Projekten, welche durch neue Ansätze eine Steigerung der Wirtschaftlichkeit, Effizienz und Genauigkeit der wohl erprobten Verfahren zu erreichen versuchen. Das in diesem Dokument behandelte Vorhaben *Adaptierung und Erweiterung von Casulli-Algorithmen für Parallelrechner mit Hardware-Beschleunigung und zur Anwendung von konservativen Advektionsverfahren* (A39530270001) ist hiermit mit einer Reihe anderer FuE-Projekte durch identische Motivation verbunden. Vor allem mit dem Projekt *UnTRIM SubGrid-Topografie* (A39550370150) [45] zusammen mit *Weiterentwicklung der Methoden zur Analyse von Simulationsergebnissen* (A39550370200), beide Abt. K, und, zusammen mit der Universität Delft und Deltares, *Effizienz- und Genauigkeitssteigerung der Modellierung der Hydrodynamik der Flüsse mit einem kombinierten Multigrid- und SubGrid-Ansatz* (A39530270002).

Zu beachten: Das in diesem Dokument behandelte Vorhaben wurde 2009 initiiert, etwa ein Jahr nach der für die Entwicklung der GPU-Programmierung entscheidenden Freigabe der CUDA Programmier-Schnittstelle durch die Nvidia Corporation, noch in einer Zeit, als numerische Berechnungen auf *Gaming*-Grafikkarten noch nicht in breiten technisch-wissenschaftlichen Kreisen ernstgenommen wurden. Das Vorhaben endet 2013, somit in einem Jahr, in welchem die Top-Maschinen aus der Liste der 500 leistungsfähigsten Rechner der Welt [70] ihre Position vor allem GPUs zu verdanken haben und GPGPU, *General Purpose Computing on Graphics Processing Units*, bereits gefragter ist, als jede andere Art des Hochleistungsrechnens – und zwar unter Verwendung preisgünstiger, handelsüblicher und massenproduzierter Hardware.

1.2 Fragestellung und Stand des Wissens

1.2.1 Entwicklung relevanter Rechnerarchitekturen

Simulation geophysikalischer Phänomene beruht auf mathematischen konzeptionellen Modellen, die aus Systemen von partiellen Differentialgleichungen bestehen. Ihre Behandlung in einem numerischen Algorithmus, der auf Netz-Methoden basiert, erfordert das Lösen großer linearer oder nicht-linearer Gleichungssysteme. Aus der rechnerischen Sicht soll also

eine große Anzahl von arithmetischen Operationen mit Vektoren und Matrizen durchgeführt werden, mit dem Verhältnis der Gleitkomma-Operationen pro Speicherzugriff (arithmetische Intensität) von 1:1. Hierzu sind Computer-Architekturen, die parallele Verarbeitung von langen numerischen Datenvektoren mit einer größeren Speicherbandbreite erlauben, am besten geeignet. Konsequenterweise führte diese Tatsache ca. 1980-2000 im Bereich des Hochleistungsrechnens (HPC, *high-performance computing*) zur Dominanz durch die ausschließlich für intensive numerische Berechnungen spezialisierten Vektorrechner. Trotz ihrer klaren Vorteile wurden diese teuren und spezialisierten Maschinen ca. 1995-2005 systematisch verdrängt und durch massiv-parallele Rechner mit gemeinsamen oder verteilten Speichern ersetzt. In den ersten Clustern dieser Art wurden immer noch spezifisch für numerische Berechnungen spezialisierte serielle Prozessoren bevorzugt verwendet. Mit dem Erscheinen von relativ preisgünstigen Mehrkernprozessoren, seit ca. 2005 ist die dominierende HPC-Architektur ein Cluster geworden, welcher aus leicht konfigurierbaren und standardisierten Rechen-Knoten besteht. Dieser Typ von Maschinen mit verteiltem Speicher basiert auf der Massenproduktion und auf weitgehend – abgesehen vom internen Netzwerk – Standard-Hardware, die auch bei PCs anzutreffen ist. Während des letzten Jahrzehnts konnte man tatsächlich einen Rückgang in der Produktion von Maschinen exklusiv für wissenschaftliches Hochleistungsrechnen beobachten, vor allem verglichen mit dem enormen Anstieg in der Menge und Qualität der Allzweck-Hardware, stationär und insbesondere mobil.

Allerdings wird die Spitzenleistung einer modernen Standard-CPU, nämlich eines MIMD-Prozessors (*Multi Instruction, Multiple Data*), für Operationen mit dem Verhältnis zwischen Gleitkomma-Operationen pro Speicherzugriff bei einem Wert von 10:1 erreicht – und das macht diese Art des Prozessors nicht die ideale Wahl für intensive arithmetische Operationen auf Vektoren und Matrizen. Die heutigen großen Mikroprozessor-basierten Cluster sind oft nicht in der Lage, die Parallelität feiner als eine Sub-Domäne (ein Teil von Daten pro Prozessor) auszunutzen. Zusätzlich und insbesondere bei Mehrkernprozessoren, wird die parallele Ausführung durch die gemeinsame von allen Prozessorkernen genutzte Speicher-Bandbreite und von der Speicher-Latenz limitiert. Die Bandbreite der in Clustern verwendeten internen Netzwerke steigt auch sehr langsam. Dazu stagniert seit Jahren die so oft als garantiert betrachtete Zunahme der Taktfrequenz der Prozessoren aus physikalischen Gründen.

1.2.2 Accelerators

Diese für numerische Berechnungen umständliche Situation wird durch Versuche mit der Anwendung von speziellen Beschleunigern (*Accelerators*) oder Co-Prozessor-Chip-Designs gemildert. Diese spezielle Einheiten sind vor allem dem Ausgleich von Schwächen derzeitiger CPUs, d.h. der Beschleunigung von Operationen auf langen Datenvektoren, gewidmet. Zu diesen Beschleunigern gehören rekonfigurierbare FPGA (*Field-Programmable Gate Arrays*), Cell-BE (*broadband engines*), bzw. auch die in der letzten Zeit größer gewordenen SIMD (*single instruction, multiple data*) Einheiten der modernen seriellen CPUs.

Doch erst die enorme Erhöhung der Leistung der überall verfügbaren und massenproduzierten GPUs (*Graphic Processing Units*), die in den letzten zehn Jahren stattgefunden hat, haben den Durchbruch in der Anwendung von Accelerators mit sich gebracht. Dies ist insbesondere der Tatsache zu verdanken, dass das Programmieren von GPUs, vorher nur einem engen Spezialisten-Kreis vorbehalten, seit der vollständigen Freigabe ca. 2007-8 von den Programmier-Schnittstellen auch für allgemeine ingenieurwissenschaftliche Zwecke möglich geworden ist. Seit dieser Zeit steigt das Potential der *Streaming*-Prozessoren von GPUs als Hardware der Wahl für rechenintensive parallele Anwendungen – vor allem für den an der Rechenleistung interessierten Personenkreis ohne Zugang zu großen Clustern. Ein weiterer Aspekt ist die Wirtschaftlichkeit der intensiven numerischen Berechnungen – GPUs liefern bei intensiven Berechnungen mehr Leistung in FLOP/s pro Watt als CPUs.

Der stetige Übergang von der homogenen zur heterogenen Rechnerarchitektur findet besonders intensiv seit ca. 2008 statt – und zwar im sehr breiten Spektrum – von großen HPC-Servern bis zu preisgünstigen Workstations. Die bisherigen spezifischen Hardware-Lösungen in der relativ elitären technisch-wissenschaftlichen Nische wurden in der letzten Zeit durch Entwicklungen im Bereich *General Purpose Computing on Graphics Processing Units*, kurz GPGPU, ergänzt und schließlich durch die Anwendung der weit verfügbaren handelsüblichen Hardware dominiert. Hiermit sind neue Wege offen, die in wissenschaftlichen Verfahren zur Simulation von physikalischen Prozessen – obwohl im Detail auf eine unterschiedliche Art – die effektive Nutzung der heterogenen Parallelität in numerischen Verfahren auf der fein-, mittel- und grobkörnigen Ebene ermöglichen, Absatz 1.2.3.

Hiermit geht auch die Zeit zu Ende, in welcher ohne grundlegende technologische Veränderungen bei der Hardware die weitere Beschleunigung sequentieller oder nur mit MPI parallelisierter Codes ohne größere Anpassungen der Software-Struktur möglich war. Die Beschleunigungen resultierten in der Vergangenheit oft nur wegen Erhöhungen der Taktrate neuer Prozessoren, die zur Zeit nicht mehr erhöht werden kann. Die Bandbreite der in Clustern verwendeten Netzwerke steigt ebenfalls sehr langsam. Eine Folge davon ist, dass die Software jetzt flexibel an die neue heterogene Parallelität angepasst werden muss. Während dies für die allgemeine Benutzer-Software ein schwieriger Paradigmenwechsel ist, sind viele Codes aus dem Bereich der physikalischen Simulation bereits vor Jahrzehnten, ihrer Natur entsprechend, so strukturiert worden, dass die vorhandene feinkörnige Parallelität mit vergangenen Vektor-Prozessoren stark beschleunigend wirkte.

1.2.3 Drei Arten der Parallelität

Der heutige Programmier einer parallelen Maschine beschäftigt sich mit drei Arten von Parallelität in numerischen Algorithmen, die zur Zeit unter Verwendung spezifischer (heterogener) Hardware am besten behandelt werden können:

- Feinkörnig – die zahlreichen spezialisierten Prozessoren einer GPU bzw. eines anderen Accelerators oder Co-Prozessors betreffend — schnelle Operationen auf Datenvektoren. Dazu werden spezialisierte Programmiersprachen verwendet, wie z.B. CUDA C (*Compute Unified Device Architecture*) [48, 59, 23], OpenCL [41], oder Direktiven in bis-

her existierenden allgemeinen Programmiersprachen, wie die Accelerator-Technologie [55] und OPENACC [49] für C/C++ und FORTRAN.

- Mittelkörnig -– heterogene Ressourcen innerhalb eines Rechen-Knotens mit gemeinsamen Speicher, d.h. zahlreiche Prozessorkerne ergänzt durch Vektor-Register (bei auffällig kürzeren Vektorlängen als in GPUs), L2-Cache -– große Flexibilität, aber von der Verfügbarkeit der spezialisierten Entwicklungswerkzeugen abhängig. In diesem Bereich gibt es die meiste Freiheiten in der Programmierung, z.B. mit der Anwendung von der klassischen seriellen Programmierung, Threading, bzw. OPENMP [50, 35]. Dazu kann man auch die Vektoranweisungen im Befehlssatz für eine SIMD-Einheit der CPU unter Anwendung von einem speziellen Compiler zuordnen (z.B. Intel AVX).

- Grobkörnig – Nutzung von Clustern bestehend aus mehreren getrennten Rechen-Knoten, verbunden mit einem möglichst effizienten Netzwerk (und dann meistens mit verteiltem Speicher). Programmier-Paradigmen basieren hauptsächlich auf der Datenparallelität (z.B. Gebietszerlegung bei Netz-Methoden) und der MPI-Kommunikation (*message-passing*) [47].

Während die verfügbaren Entwicklungswerkzeuge in der Anfangsphase hier beschriebenen Vorhabens (d.h. 2009) die Arbeit mit diesen drei Arten der Parallelität *getrennt* erlaubten, gehörte eine Kombination von denen in einem Hybrid-System gehörte zu den Fragestellungen der (meist industriellen) Forschung. Heute (2013) ist dies immer noch relativ umständlich und abhängig von dem Anwendungsfall, aber bereits die gängige Praxis auf heterogenen Clustern.

1.2.4 Manycore-Co-Prozessoren und APUs

Die immer noch vorhandenen Probleme auf der feinkörnigen Ebene sind vor allem durch derzeitige Einschränkungen der Hardware bezüglich der Datentransfers über den PCIe-Bus, d.h. zwischen dem Host (CPU-Kerne) und den Geräten (*devices*, z.B. eine einzelne oder mehrere GPUs) bedingt, oder genauer gesagt, zwischen den von diesen Einheiten kontrollierten, physisch getrennten Speichern. Darüber hinaus ist die mehrstufige Kommunikation zwischen den Geräten in unterschiedlichen Rechen-Knoten eines Clusters über das externe Verbindungsnetzwerk eine weitere Limitierung.

Diese Probleme sind auch den Hardware-Produzenten bewusst. Dies hat dazu geführt, dass eine neue Technologie seit etwa 2011 zur Verfügung steht, nämlich die APU (*Accelerated Processing Unit*)-Technologie, Verschmelzung von CPU und GPU auf einem Chip, mit gemeinsamen Caches und den Zugriff auf denselben Speicher – was die Latenzzeit bei der Datenübertragung verringert und radikal das Speicher-Management vereinfacht. Die am weitesten verbreiteten Einheiten sind zur Zeit die energieeffizienten Prozessoren für mobile Geräte, wie AMD *Fusion* oder sind noch in der Entwicklung, wie Nvidia *Project Denver*.

Parallel zu dieser Entwicklung ist 2012 nach vielen Anläufen in den letzten Jahren der erste allgemeine *manycore*-Prozessor (eine Steigerung zum Begriff *multicore*) samt Entwicklungswerkzeugen auf dem Markt erschienen, nämlich Intel *Xeon Phi* (bekannt auch als MIC

bzw. früher unter den Code-Namen *Knights Corner* oder *Larrabee*), in der Form einer über die PCIe-Schnittstelle mit dem Gesamtsystem verbundenen getrennten Karte. Xeon Phi soll in einem Chip eine größere Anzahl (mehr als 50) von seriellen x86-Prozessoren mit erweiterten Vektorregistern, einem schnellen Cache und einer sehr guten Anbindung an den eigenen Speicher (zur Zeit bis 8GB) vereinen. Dieser Prozessor wird als universeller (Co-)Prozessor für breitbandige graphische und rechnerische Aufgaben gesehen, der zusätzlich auch in der Lage sein sollte, die Aufgaben von bisherigen Host-CPUs zu übernehmen (z.B. das Betriebssystem). Vor allem ist er aber als ein paralleler Co-Prozessor gedacht, der durch einen deutlich umfangreicheren Befehlssatz als bei GPUs für komplexere Aufgaben universeller einsetzbar ist. Der Preis dafür ist eine um zwei Größenordnungen kleinere Anzahl der Prozessoren.

Auf jeden Fall werden Applikationen, die durch Vektorisierung und Optimierung der Speicherbandbreite auf GPUs beschleunigt werden können, auch auf den *Xeon Phi* Co-Prozessoren bzw. APUs erfolgreich angewendet, da genau diese gleichen fundamentalen Eigenschaften in allen Fällen für die effektive Nutzung der Hardware kritisch sind.

1.2.5 GPU-Architektur

In diesem Absatz werden die Eigenschaften einer GPU genauer beschrieben, betrachtet als ein Co-Prozessor für numerische Berechnungen und mit der speziellen Aufmerksamkeit auf eine Nvidia Fermi GPU – da dies der Accelerator der Wahl in diesem Projekt ist.

Ein Accelerator (Beschleuniger) wird typisch in der Form eines Co-Prozessors zum Host-Prozessor (CPU) verwendet. Er hat einen eigenen Befehlssatz und üblicherweise (aber nicht immer) auch einen eigenen Speicher. Für das gesamte Computersystem sieht der Beschleuniger wie eine Eingabe/Ausgabe-Einheit aus: Er kommuniziert mit (Host-)CPU über die PCIe-Schnittstelle mit I/O-Befehlen und mit DMA-Transfers (DMA: *Direkt Memory Access*). Für die Software, ist der Beschleuniger ein anderer Rechner, zu dem die Programme Daten und Routinen zur Ausführung senden können. Mit aktuellen Technologien werden Accelerators genauso wie die CPU als einziger Chip produziert, wie Intel MIC, AMD APU, Cray FPGAs und schließlich GPUs. Im Weiteren konzentrieren wir uns auf die Nvidia GPUs; ein Blockdiagramm mit den für das Programmieren relevanten Anteilen einer Fermi GPU wird in Abb 1 dargestellt.

Die wichtigsten Bestandteile einer GPU sind *Streaming*-Prozessoren, hierarchischer Speicher und die Verbindungen zwischen ihnen. Nvidia Fermi GPUs haben bis zu 16 Multiprozessoren; jeder Multiprozessesor hat zwei SIMD-Einheiten mit je 16 parallelen *Stream Processing Units* (Thread-Prozessoren, *Shader*). Thread-Prozessoren arbeiten auf synchronische Weise, d.h. alle Prozessoren in einer SIMD-Einheit führen den gleichen Befehl in der gleichen Zeit aus. Multiprozessoren arbeiten dagegen asynchronisch, genauso wie Prozessorkerne einer üblichen CPU.

Die GPU hat einen eigenen Speicher (*device memory*), zur Zeit bis zu 6GB. Genauso wie bei CPUs sind die Zugriffszeiten zum Speicher im Vergleich zur Ausführungsgeschwindigkeit relativ langsam. Um die mit der Speicherlatenz verbundenen Effekte zu mildern, wer-

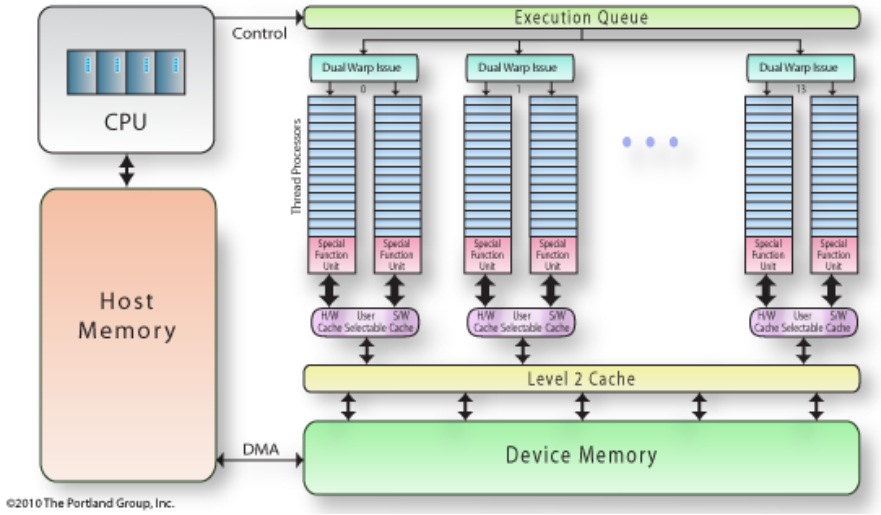

Abbildung 1: *Blockdiagram von der Nvidia Fermi GPU betrachtet als Beschleuniger, Bildnachweis: PGI-Insider, Ausgabe März 2012, Portland Group.*

den bei CPUs schnelle interne Speicher verwendet, die für numerische Berechnungen die Rolle von kontrollierbaren Level-2-Caches spielen. In den Caches werden die letztlich verwendeten Datenbereiche zwischengespeichert in der Hoffnung, dass die nächsten Zugriffe aus dem Cache bedient werden können. (Falls das nicht zutrifft, spricht man über sog. *cache-misses* bzw. *-stalls*.) Caches sind ziemlich effektiv, insbesondere bei der Cache-optimierten Programmierung, jedoch lösen sie die prinzipiellen Probleme nicht, die alle heutigen Prozessoren mit der limitierten Speicherbandbreite und relativ großen Speicherlatenz haben. Die numerischen GPU-Programme benötigen üblicherweise fließende Zugriffe zu größeren Datensätzen und zwar in solchen Mengen, dass jede übliche Cache schnell überfüllt sein würde. Um dieses Problem zu lösen, verwenden GPUs Multi-Threading. Falls ein GPU Prozessor einen Befehl zum Speicherzugriff ausgibt, wird dieser GPU Thread inaktiv bis der Speicher die benötigten Datenwerte liefert. In der Zwischenzeit schaltet der GPU-Prozessor schnell (d.h. in der Hardware) zu einem anderen Thread um und führt diese aus. Auf diese Weise nutzt die GPU die feinkörnige Parallelität im Programm, um ständig beschäftigt zu werden, während der langsame Speicher auf die Anforderung antwortet und die benötigten Daten liefert.

Während der interne Speicher einer GPU eine gewisse Latenz hat, hat die Verbindung zwischen dem Speicher und den GPU-Prozessoren eine beachtliche Bandbreite, um ein Vielfaches größer als bei einer CPU anzutreffen ist. Im Gegensatz zum von der CPU kontrollierten Hauptspeicher, kann der interne GPU-Speicher mit den hohen Anforderungen von datenintensiven Programmen mithalten. Anstatt von den *cache-misses* betroffen zu werden, arbeitet GPU weiter – vorausgesetzt, dass es genug parallel zu bearbeitende Daten gibt, um die Thread-Prozessoren ständig zu beschäftigen.

1.2.6 Programmieren von GPUs

Unter der Verwendung von Verwendung von Nvidia CUDA [48] bzw. der Open-Standard Programmiersprache OpenCL [41] haben sich zur Zeit jeweils zwei Ansätze der Programmierung von GPUs durchgesetzt. Während CUDA die erste Wahl für Programmier ist, die Geschwindigkeitssteigerungen ihrer Applikationen spezifisch auf Nvidia Grafikkarten suchen, spezialisiert sich OpenCL auf die Portabilität für andere GPUs und Acceleratoren von verschiedenen Produzenten und mit diversen Philosophien.

Die zusätzlichen Kosten bei der Programmierung unter Verwendung von CUDA oder OpenCL sind vor allem durch den anfänglichen Aufwand gegeben, einen sonst für eine homogene Rechenarchitektur einspurigen Code in zwei Teile, einen für die CPU (*host*) und den zweiten für die GPU (*device*), zu konvertieren. Jede Routine muss in eine separate *Kernel*-Funktion extrahiert werden, um auf dem Accelerator ausgeführt werden zu können. Der Host-Code muss die Speicherverwaltung auf dem Host und auf dem Gerät, explizite Datentransfers und Aufrufe der Kernel-Funktionen verwalten. Die Kernel-Funktionen müssen sorgfältig für die gegebene GPU-Architektur konfiguriert werden (Abb. 8). Während CUDA und OpenCL eine detaillierte Kontrolle der Programmabläufe und sehr gute Optimierungen erlauben, sind sie im Allgemeinen nicht einfach zu programmieren. Mit diesen Ansätzen zu experimentieren ist ohne spezifische Einarbeitung nicht empfehlenswert.

Deswegen gibt es auch weitere Entwicklungen, die auf allgemeinen Programmiersprachen basieren und eine Hardware-Adaptierung eines Codes mittels einer API (*Application Programming Interface*) ermöglichen. Sie bestehen aus Direktiven, die man in einem existierenden seriellen Code streut, und einem Satz von API-Routinen für spezifische Aufgaben (genauso wie OpenMP [50]). Es handelt sich hier im Vergleich zu den o.g. Sprachen, die bereits eine gewisse Reife aufweisen, um eine immer noch laufende Entwicklung. Sie hat in der chronologischen Reihenfolge mit der Accelerator-Technologie von PGI [55] für CUDA und der allgemeineren OpenHMPP-Technologie von CAPS [69] angefangen. Die 2011 erschienene OpenACC-Technologie [49] wird oft als Nachfolger der beiden dargestellt. Schließlich wird erwartet, dass 2013 eine Verschmelzung all dieser Technologien (auch für div. andere als nur GPU Accelerator-Arten) in der Form von OpenMP 4.0 [51] stattfinden wird.

Mit dem Direktiven-gesteuerten Ansatz ist der Einstieg in die GPU-Programmierung viel einfacher und erlaubt Experimente, mit denen man die Aussichten einer Portierung existierender Codes zu GPUs schnell beurteilen kann. Diese Entwicklung scheint besonders für diese traditionell denkende Programmierer attraktiv zu sein, die größere *legacy*-Codes mit möglichst kleinem Programmieraufwand und wenigen Verlusten an der allgemeinen Portierbarkeit für GPUs bzw. auch andere Acceleratoren adaptieren möchte. Dies scheint bereits mit der OpenACC-Technologie machbar zu sein, jedoch immer noch mit relativen Einbüßen in der rechnerischen Performance, die zur Zeit eher nur mit hoch optimierenden Sprachen, wie CUDA, zu erreichen ist.

Genaue technische Spezifikationen der in diesem Projekt verwendeten Grafikkarte werden in Kap. 4.2 erörtert. Die benutzten Programmier-Ansätze werden in Kap. 1.4.3 und 3.2 dargestellt.

1.2.7 TRIM und UnTRIM

Eins der Verfahren, welches vor über einem Jahrzehnt durch Eigenentwicklungen der BAW und hochwertige Auftragsarbeiten (Cray, SGI) an die Architektur eines parallelen Vektorrechners bestens angepasst und weiterentwickelt wurde, ist das Finite-Differenzen/Volumen Verfahren TRIM-2D bzw. TRIM-3D. Im Kontrast zu dem Nachfolge-Programm UNTRIM arbeiten sie mit kartesischen strukturierten Netzen. Neben dem klaren Potential für die erfolgreiche Nutzung der heterogenen Rechnerarchitektur unter Verwendung von Acceleratoren spricht für ein Verfahren mit kartesischen Netzen die breite Verfügbarkeit von hoch aufgelösten topographischen Daten in der Raster-Form, Absatz 1.2.8. Zusätzlich sind kartesische Netze sehr vorteilhaft für Experimente mit neueren numerischen Verfahren, die für diese Art der Netze die besten Eigenschaften aufweisen. Gegen Entwickungen für strukturierte Netze spricht die Tatsache, dass die Weiterentwicklung von TRIM zugunsten vom Nachfolger UNTRIM (unstrukturierte orthogonale Netze) ab etwa 1999 praktisch aufgegeben wurde, was eine Neu-Implementierung von TRIM mit sich bringen kann, um alle neuen Entwicklungen einzubeziehen.

UNTRIM ist durch optimierte Cache-Nutzung, OPENMP-Parallelisierung (Pallas [54], mit anschließenden Arbeiten von Prof. Casulli und BAW) und schließlich MPI-Parallelisierung (BAW) bestens an die homogene (d.h. auch *multicore*) Rechnerarchitektur angepasst. Die Optimierung für alle Ebenen der heterogenen Parallelisierung ist mit nicht unerheblichen Umstrukturierungen im Code verbunden (insbesondere für die Operationen auf Datenvektoren) und von der Verfügbarkeit bzw. Bereitstellung der fortgeschrittenen Entwicklerwerkzeugen abhängig, insbesondere für die *manycore*-Co-Prozessoren, aber durchaus denkbar.

Während die Anpassung von UNTRIM für die neuen Programmier-Paradigmen erstrebenswert ist, hat TRIM als Pilotverfahren größere Erfolgschancen auch aus dem Blickwinkel der Weiterentwicklung grundlegender numerischer Verfahren (Advektion, Abschnitt 1.2.9). Die wichtigsten von algorithmischen Veränderungen seit 1999 (nicht-lineares Verfahren für Überflutung und Trockenfallen, Subgrids) könnten optional in TRIM re-implementiert werden. Schließlich ist auch eine komplette Neu-Implementierung des grundlegenden Algorithmus von TRIM, insbesondere im zweidimensionalen, vertikal gemittelten Fall schnell zu erreichen.

1.2.8 Kartesische Gitter

Eine zur Zeit unverzichtbare Hilfe bei der Erstellung von ortspezifischen Modellen sind Raster-basierende, digitale Gelände-Modelle (DGM). Insbesondere bei Anwendungen für den Hochwasserschutz ist ihre Präzision und die Leichtigkeit der Aktualisierung sehr hoch geschätzt. Moderne Technologien, wie z.B. Lidar, liefern DGMs mit stetig steigender Genauigkeit, die Bearbeitungszeiten zur Erstellung von Datensätzen, die direkt in der Modellierung verwendbar sind, immer kürzer.

Kartesische (strukturierte) Gitter haben offensichtlich viele Vorteile im Vergleich zu unstrukturierten Netzen. Vor allem ist die Netz-Generierung mit der Verwendung von Raster-basierenden DGMs sehr schnell und kann vollständig automatisiert werden. Dies betrifft

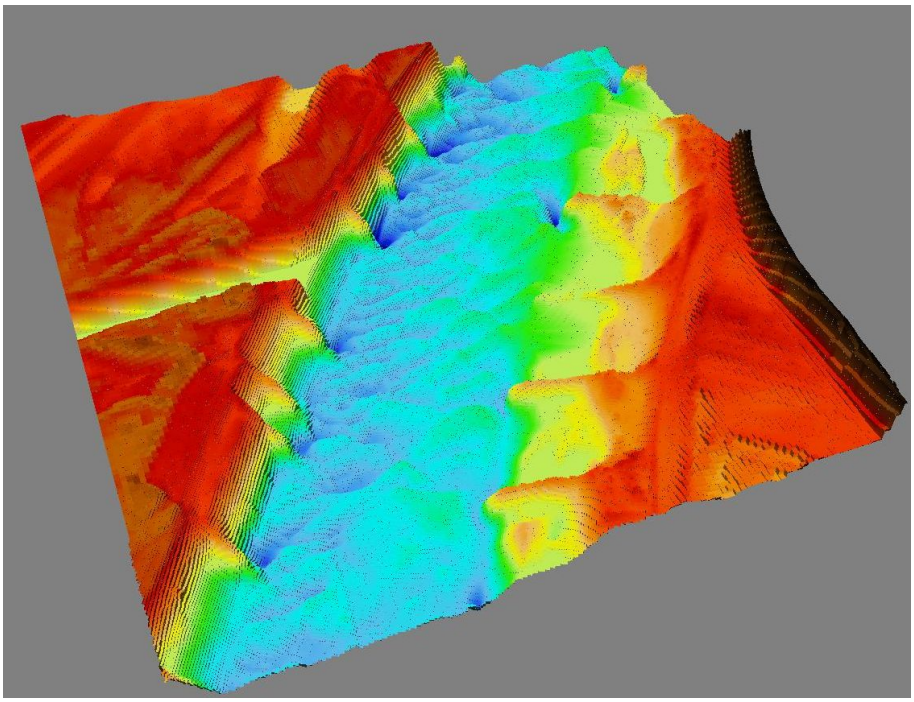

Abbildung 2: *Ausschnitt des* UnTRIM*-Modells Die Elbe bei Hitzacker. Obwohl das Netz in diesem ortspezifischen Modell unstrukturiert ist, wirkt es wegen der feiner Auflösung wie ein feines Finite-Differenzen-Gitter.*

auch die gezielte, Kriterien-gesteuerte Verfeinerung, Vergröberung und die Netzadaptivität während der Simulation. Diese Einfachheit der direkten Übertragung von DGMs sofort in das kartesische rechnerische Netz wurde bereits sehr früh bemerkt und insbesondere bei fortgeschrittener Hochwassermodellierung verwendet [53, 36, 61].

Viele von den oft erwähnten typischen Nachteilen der kartesischen Netze – insbesondere bei der Erfassung von Berandungen und der Küstenlinie – wurden durch die Einführung von neuen Methoden gemildert, bzw. entfernt, z.B. *Cut Cell Method* [57, 58], *Immersed Boundary Method* [20], *Quadtrees* [61, 67], *SubGrids* [16, 45, 61]. Bei der Verfolgung der Schnittstelle Land/Wasser, die sich horizontal bewegt, wie bei von Gezeiten beeinflussten Wattflächen, überfluteten Vorländern von Flüssen, Dammbrüchen, etc. haben alle Netzarten Probleme, die auch auf ähnliche Weise und unabhängig von der Netzart gelöst werden können, z.B. [26, 11]. Nur bei steile(re)n Berandungen (Kaimauer, Deiche), die sich mit variierenden Wasserständen im Raum nicht bewegen, sind unstrukturierte Netze vielleicht besser geeignet.

In der Abb. 2 wird ein Ausschnitt des Netzes für das UnTRIM-Modell *Die Elbe bei Hitzacker* gezeigt, erzeugt für alle Wasserstände von Niedrig- bis Hochwasser, d.h. die Uferlinie wird innerhalb des Netzes durch das Verfahren selbst bestimmt. Obwohl das Netz in diesem ortspezifischen Modell unstrukturiert ist, schwinden bei der sehr feiner Auflösung $O(2\mathrm{m})$ die Unterschiede zwischen einem unstrukturierten Netz mit Vierecken und einem kartesischen Netz – genauso, wie die Vorteile der Verwendung von unstrukturierten Netzen.

1.2.9 Advektive Prozesse

Eine Schwäche von allgemeinen Verfahren für die numerische Modellierung der natürlichen Gewässern ist, dass die Simulation der Übergänge zwischen schießender und strömender Strömung oft fehlerbehaftet ist. Dieser Fehler ist prinzipiell von der Art abhängig, wie die Advektionsterme in diesen Verfahren numerisch behandelt werden. Allgemeine Verfahren zu langfristigen Simulationen nutzen vor allem diese Advektionsverfahren, die zwar die Massenerhaltung gewährleisten, aber nicht immer die gleichzeitige Erhaltung des Impulses und der Energie. Dadurch werden die Strömungswechsel oft nicht richtig reproduziert.

Für mesoskalige Modellierung existieren pragmatische Ansätze zur Lösung dieses Problems, die in TRIM und UNTRIM implementierbar sind, die sich abhängig von der Veränderung der Froude-Zahl entlang einer Stromlinie an die Energie- oder Impulserhaltung umstellen [44, 62].

Außerdem gibt es eine Reihe weiterer Ansätze aus der Domäne der advektionsdominierten Strömungen (nach der Theorie der hyperbolischen Gleichungen), die aber für unstrukturierte Netze und in drei Dimensionen mit der Anwesenheit der freien Oberfläche Bestand der laufenden Forschung sind. Für strukturierte kartesische Netze existieren bereits mehrere Adektionsverfahren, die vielversprechend sind [65].

Da für die typischen Wasserstraßen die o.g. Übergänge im Typ der Strömung nur bei der Überströmung von Bühnen bzw. Parallelwerken, Dämmen, Böschungen stattfinden, sind sie auf vorhersehbare Weise auf bestimmte und kleine Bereiche des Gesamtmodells begrenzt. Um die Wirkung von Wasserbauwerken auf die Wasserstände bei sehr variablen Hochwasserverhältnissen und Bodentopographie möglichst genau zu reproduzieren, wurden bereits dreidimensionale nicht-hydrostatische Modelle angewendet.

Wegen der Lokalität der Übergänge im Typ der Strömung wird die Anwendung von vollständigen hyperbolischen *shock-capturing*-Modellen als prinzipiell unwirtschaftlich bzw. problematisch gesehen. Diese Godunow-, Riemann-, bzw. Roe-Methoden wurden für die möglichst genaue Erfassung der Ausbreitung von Schock-Wellen im ganzen Modell-Bereich entwickelt, wie z.B. bei Dammbrüchen, schnell verlaufende Hochwässern, Wellen [64]. Wegen Stabilitätskriterien und der Numerik hyperbolischer Gleichungen wird die Nutzung dieser Modelle bei sehr stark veränderlichen Wassertiefen und quasi-stationären Zuständen erschwert. Die Erweiterung dieser Methoden auf drei Dimensionen und die Berücksichtigung der vertikalen Beschleunigungen ist immer noch problematisch für Strömungen mit freier Oberfläche.

Anmerkung: Die vereinfachten elliptischen Modelle aus dem wasserwirtschaftlichen Bereich (*diffusion wave, storage-cell*), die die Advektion vernachlässigen [36] scheinen zur verantwortlichen Beurteilung der Auswirkung von Wasserbauwerken auf die Wasserstände prinzipiell ungeeignet zu sein, insbesondere bei hoch aufgelösten Modell-Topographien.

1.3 Bedeutung für die WSV

Wegen der schwerwiegenden Veränderungen im Bereich Hochleistungsrechnen werden die numerischen Verfahren nicht mehr automatisch schneller bzw. effizienter einsetzbar nur durch die Beschaffung von neuer Hardware. Sorgfältige Anpassung und Weiterentwicklung der Software ist daher als ein – im Vergleich zu Hardware – langfristig wirkender und Kompetenz-entwickelnder Forschungsbeitrag anzusehen. Neue Rechnerarchitekturen -– bei einer erfolgten Anpassung der mathematischen Software — erlauben zusätzlich sehr energieeffizientes Rechnen im Sinne der erzielten rechnerischen Leistung pro Watt verbrauchter Leistung — ein evidenter Vorteil bei der Wirtschaftlichkeit des Hochleistungsrechnens.

Die immer detaillierteren Prognose-Anforderungen der WSV, auch in Bezug auf kleinräumige Effekte, setzt den Einsatz extrem fein diskretisierter Modellgebiete und besonders die exakte Modellierung der Wasserwechselzonen mit Bauwerksüberströmungen voraus.

Eine Erweiterung der Einsatzmöglichkeiten der wohl-validierten und typischen Verfahren für mesoskalige (d.h. z.B. Flussstrecken von einigen Kilometern Länge) und langfristige (mehrere Stunden bzw. Tage) Modellierung auf die korrekte Erfassung der Übergänge im Strömungscharakter, insbesondere Schießen/Strömen (sehr variable Froude-Zahlen entlang einer Stromlinie), kann die Prognosefähigkeit dieser Verfahren wesentlich erhöhen. Hiermit können die Wasserstände in der Nähe von gerade überströmten Wasserbauwerken (wie Buhnen, Parallelwerke, etc.) sehr korrekt berechnet werden, was tatsächlich klare Vorteile bei der sicheren Beurteilung der Hochwasserneutralität dieser Bauwerke bei allen Wasserständen mit sich bringt. Die Abarbeitung mehrdimensionaler Feststofftransportmodelle kann erheblich beschleunigt werden.

1.4 Vorgehensweise

1.4.1 Grundgedanke

Da die Leistungsfähigkeit von neuer hybrider Hardware, wie APUs und MICs als Co-Prozessoren in traditionellen CPU-basierenden Rechen-Knoten, im Vergleich zu den neuen GPU Designs noch bewertet werden muss, besteht weiterhin eine gewisse Unsicherheit, wie man die bestehenden CFD-Codes an die sich schnell ändernde Hardware-Architektur anpasst.

Trotz dieser derzeitig zu beobachtenden Mehrdeutigkeit in den Hardware-Entwicklungen wird es zu dem jetzigen Zeitpunkt empfohlen, in Arbeiten zu investieren, die die feinkörnige Parallelität in den vorhandenen Algorithmen hervorheben und nutzen. Die Begründung dafür ist, dass die Hauptquelle der rechnerischen Leistung der vorliegenden GPUs und kommenden APUs/MICs die Hardware-beschleunigte parallele Operationen auf Datenvektoren sind. Deswegen wird als Hauptziel dieser Studie nicht ein Versuch sein, das gesamte Spektrum der derzeit verfügbaren heterogenen parallelen Ressourcen zu verwenden – wie die Berechnung in einem Tandem von CPU und GPU oder Anwendung mehrerer GPUs. Stattdessen konzentrieren wir uns auf die grundlegende Vorgehensweise, die auch den CUDA-Ansatz auszeichnet.

In diesem Ansatz nutzt man die CPU vor allem für die Steuerung des Rechenablaufs während möglichst die gesamte rechnerische Hauptlast des Codes für die parallele Ausführung auf eine GPU gebracht wird. Statt einer Anpassung eines bestehenden seriellen, mit OPENMP- oder MPI-parallelisierten Codes auf eine GPU mit allen notwendigen Kompromissen, die zu erfüllen sind, wird ein Rechenkern – der wichtigste Algorithmus eines vertikal integrierten Gleichungslösers für allgemeine Flachwasserströmungen – *vollständig* und sorgfältig für eine GPU in einer spezialisierten Programmiersprache neu implementiert. Es handelt sich also um den grundlegenden Algorithmus des Programms TRIM-2D, erweitert um die nicht-lineare Betrachtung von Überflutungen. Mit vereinfachten Anfangs- und Randbedingungen, aber mit allen typischen Merkmalen des grundlegenden 2DV-Schemas nach Casulli und der zu bewältigenden Rechenlasten. Dieser konzentrierte, radikale Ansatz ermöglicht eine klare Beurteilung, welche Geschwindigkeitssteigerungen zu erwarten sind, wenn ein nicht-trivialer Gleichungslöser für Flachwasserströmungen für eine *state-of-the-art* GPU vollständig und mit den besten derzeit verfügbaren Programmier-Tools und Bibliotheken implementiert wird.

1.4.2 Ziele

Nach den Ansichten von 2009 (d.h. am Anfang dieses Projekts), um die neuen Rechnerarchitekturen effektiv zu nutzen und Beschleunigung der Ausführungszeiten zu erreichen, mussten die von der Abt. W und K verwendeten Verfahren sorgfältig angepasst werden. In der Vergangenheit wurden mit der zeitlichen Entwicklung die nachfolgenden Versionen dieser numerischen Verfahren an die wechselnden Rechnerarchitekturen angepasst, so dass wir in diesem Projekt über einen Fundus an Methoden und Lösungen verfügen, die erfolgreich wiederbelebt werden konnten. Vor allem betrifft das die Anpassungen für Vektorrechner und für Parallelrechner mit gemeinsamen Speicher. Die Auswahl, in welche Methoden weiterhin investiert werden soll, sollte nicht nur Performance und Wirtschaftlichkeit berücksichtigen, sondern auch Potential für zukünftige Entwicklungen und Anwendungen haben. Hiermit besteht auch die Chance, die Kernkompetenz der BAW in numerischen Verfahren für Wasserstraßen zu erhalten und weiterzuentwickeln.

Als es 2009 geplant war, hatte das Vorhaben prinzipiell drei Ziele:

1. Beherrschen der neuen Möglichkeiten im Hochleistungsrechnen auf kommenden Rechnerarchitekturen, wobei die Nutzung der heterogenen Parallelität im breiten Sinne gemeint ist. Dazu sollte ein Verfahren verwendet werden, das bereits in der Vergangenheit auf eine sehr ähnliche Art der Ausführung erfolgreich optimiert wurde und repräsentativ für eine Klasse der BAW-Verfahren ist. Dazu ist TRIM wegen der exemplarischen Einfachheit, Klarheit und Leistungsparameter auf einem parallelen Vektorrechner als Testverfahren ideal.

2. Abhängig von den Resultaten der Ersatzbeschaffung des neuen Compute Servers der Abt. W: Implementierung von TRIM auf einem Cluster bestehend aus Knoten

mit Multi-Kern-CPUs und GPUs (z.B. vorteilhaft realisiert als Teil derzeit Ersatzbeschaffenden Maschine), bzw. auf einem getrennten Rechen-Knoten (ein PC mit CPU+GPU und geeigneten Entwicklerwerkzeugen). Im ersten Fall soll TRIM auch mit MPI und einer einfachen Gebietszerlegung für strukturierte Netze grobkörnig parallelisiert werden. Die gewonnenen Erfahrungen sollen auch für andere Verfahren übertragbar sein, insbesondere für UnTRIM.

3. Arbeit an neuen Advektionsverfahren zur Erfassung der lokal und räumlich begrenzt auftretenden Übergänge im Strömungscharakter (d.h. zwischen schießenden und strömenden, bzw. über- und unterkritischen Abfluss), insb. bei den für Fragestellungen an Wasserstraßen typischen Froude-Zahlen variabel um den Wert 1. Die Erfassung dieser Übergänge liegt bisher teilweise außerhalb des Anwendbarkeitsbereiches für Casulli-Verfahren. Hiermit sollen die günstigen Eigenschaften der strukturierten Netze (wie in TRIM) ausgenutzt werden, da ausgerechnet für diese Art von Netzen (im Vergleich zu den unstrukturierten) eine Anzahl von versprechenden Advektion-Schemata gibt, auch höherer Ordnung. Zuerst sollen adaptive Verfahren nach Duinmeijer und Stelling [62] bzw. Kramer und Stelling [44] eingebaut werden. Eine besondere Aufmerksamkeit gilt insbesondere diesen Verfahren, die sowohl in 2D, wie auch in 3D ähnliche Eigenschaften haben sollen.

1.4.3 Die Auswahl der Untersuchungsmethoden

Als in 2009 anvisiert, sollten die Entwicklungsarbeiten abhängig von den Ersatzbeschaffungen 2010 und 2012 für die Compute-Server der Abteilung W erfolgen:

- Entweder auf dem Compute-Server, bevorzugt auf einer für Entwicklungsarbeiten geeigneten Cluster-Partition, falls sie mit GPU-Hardware ausgestattet wird
- oder auf einem getrennten Rechen-Knoten mit geeigneten GPU(s) ausgestattet, der als ein leistungsfähiger PC realisierbar wäre.

Da in der Ausstattung von sowohl dem in 2010 von SGI lieferten Compute-Server `cleopatra`, wie auch von dem in 2012 von Bull gekauften Compute-Server `brutus` keine Rechen-Knoten mit GPU vorhanden waren, erfolgten die Arbeiten 2010-12 auf einem im Rahmen der Bachelorarbeit von Marcus Brückner [6] gebauten und später freundlicherweise von der Abt. Z zur Verfügung gestellten PC mit zwei Nvidia GeForce GTX480 Grafikkarten, wovon langfristig nur eine nutzbar war. Deswegen wurden auch keine Untersuchungen für Algorithmen mit mehreren Rechen-Knoten bzw. mehreren Grafikkarten realisierbar.

Auf dem gegebenen PC-System wurden 2010 teilweise im Rahmen der o.g. Bachelorarbeit einige Vorarbeiten zur Beurteilung der Werkzeuge, die eine programmiertechnisch einfache Adaptierung des vorhandenen Codes versprachen, durchgeführt. Dies wurde wegen Vorbereitungsarbeiten für ein anderes FuE-Vorhaben exemplarisch mit dem von LNHE EDF (Electricité de France) zur Verfügung gestellten SPH-Code SPARTACUS-2D [38]

realisiert. Wegen der spezifischen *trivial parallelen* Eigenschaften der numerischen Algorithmen, die aus der SPH-Methode (*Smoothed Particle Hydrodynamics*) [66] resultieren, profitieren SPH-Codes für die Ausführung auf GPUs im besonderen Maße [31, 32, 37]. Der serielle SPARTACUS-2D-Code wurde noch vor der Beschaffung eines mit geeigneter Grafikkarte ausgestatteten Rechners sorgfältig profiliert und die rechenintensivsten Bereiche mittels OPENMP-Technologie [50] für die Ausführung auf einem Mehrkernprozessor parallelisiert. Die dabei entdeckten Vektor- bzw. parallele Abhängigkeiten [35] wurden durch Veränderungen der Datenstrukturen und einiger Rechenschleifen entfernt. Darauf aufbauend wurden anschließend im Rahmen der o.g. Bachelorarbeit ansatzweise die meistversprechenden Teile des Codes mit der OPENMP-ähnlichen Accelerator-Technologie von PGI (Portland Group) [55] zur Ausführung auf GPU adaptiert.

Die so gewonnenen Erfahrungen mit der Accelerator-Technologie [6] wurden ebenfalls durch Tests mit dem TRIM-Code bestätigt: Für kleinere Datenmengen werden die mit der Accelerator-Technologie und einer GPU erzielten Beschleunigungen zwar besser als diese, welche mit OPENMP auf CPUs erreicht wurden, sie betreffen aber nur die parallelisierbaren Anteile des Codes. Es zeigte sich auch, dass in diesem Entwicklungsstadium der Accelerator-Technologie (2010) eine entscheidende Milderung des Hauptproblems des oft auftretenden zeitaufwändigen Transfers zwischen CPU-kontrolliertem Hauptspeicher und GPU-Speicher nicht gab. Übertragen von größeren Code-Anteile auf die GPU, wie die gesamten Funktionen bzw. Unterprogrammen, oder eine Kontrolle der Speicherbelegung auf der GPU, war zu dieser Zeit nicht möglich. (Erst Entwicklungen in 2012-13 im Rahmen des OPENACC-Standards [49], bzw. den kommenden Standards OPENMP 4.0 [51] haben einige von diesen Problemen gelöst.)

Da die existierenden Codes vom TRIM und UNTRIM in FORTRAN-90 geschrieben sind, wurden auf der Suche nach mehr Flexibilität und Kontrolle über den Programmablauf exemplarische Versuche mit Matrix-Vektor-Operationen, die für TRIM typisch sind, mit PGI CUDA FORTRAN durchgeführt. Es zeigte sich aber, dass CUDA FORTRAN zwar Schreiben von größeren CUDA-Kernel für GPUs erlaubt, für viele Funktionalitäten spielt diese Sprache aber nur eine Rolle als Interface zu CUDA C. Die neuen Releases von CUDA FORTRAN von 2013 haben dies nicht verändert [56]. (Es gab auch Bedenken, ob die mit PGI FORTRAN Compiler erreichte Optimierung derjenigen von dem Intel Compiler [68] ebenbürtig ist, obwohl dieser Aspekt nicht genau untersucht wurde.)

Schließlich, wie bereits in Abschnitt 1.4.1 erwähnt, wurde eine Neu-Implementierung des grundlegenden Algorithmus unter Verwendung von CUDA C mit C++-Erweiterungen [48] angefangen. Diese Sprache ist neben dem OPENCL [41] ein Industrie-Standard für die Nutzung der GPUs geworden, mit den besten bisher verfügbaren Entwickler-Tools und vor allem mit vielen gebrauchsfertigen Bibliotheken aus dem Bereich der linearen Algebra, die Hardware-beschleunigte Operationen beinhalten. Dies wird durch zahlreiche Publikationen von erfolgreichen (Re-)Implementierungen von wissenschaftlichen Codes in den letzten Jahren bestätigt, z.B. [5, 24, 25, 30, 31, 32, 37]

1.4.4 Verlauf des Projektes

Der Verlauf des hier behandelten Projektes spiegelt die wechselhafte Einstellung in der Abteilung Wasserbau zu der eigenen technisch-wissenschaftlichen Programmentwicklung und Verwendung von hoch aufgelösten Modellen wider. Ohne Erwähnung der Veränderungen im Verlauf der Zeit sind die Abweichungen zwischen den geplanten Arbeiten und den Ergebnissen nicht verständlich.

- Das Vorhaben wurde Anfang 2009 initiiert und Mitte 2009 zur Diskussion in VBC2/3 übergeben, Beantragung erfolgte im September 2009, Genehmigung im Februar 2010.
- Erste unregelmäßige Erfahrungen mit GPUs erfolgten erst nach dem Erscheinen des Nvidia Fermi-Chips ab Mai 2010 und haben Charakter der Vorarbeiten gehabt, wie im Abschnitt 1.4.3 beschrieben. Intensivere Arbeiten am Casulli-Algorithmus selbst erfolgten nicht.
- Wegen personalen Veränderungen und zeitlichen Schwierigkeiten wurde am Anfang 2011 eine ergebnislose Abschließung des Vorhabens vorgeschlagen.
- Das Projekt wurde im März 2011 wieder mit der neuen Auflage belebt, eine klare Abschätzung der möglichen maximalen Geschwindigkeitssteigerung unter Verwendung von einer GPU bei einem möglichst repräsentativen Verfahren.
- Die neuen Auflagen bedeuteten, dass man sich auf den eigentlichen Rechenkern beschränken soll. Bis Mai 2011 wurde das für TRIM-2D grundlegende 2DV-Schema mit CUDA C vollständig neu implementiert und die geforderte Abschätzung geliefert und präsentiert.
- Wegen weiteren personalen und konzeptuellen Veränderungen verloren weitere Arbeiten an dem zweidimensionalen Code an Bedeutung und beschränkten sich bei unregelmäßigen Arbeiten Mitte 2011 – Mitte 2012 nur an der Implementierung und dem Testen von Randbedingungen, neuen Advektionsverfahren, Re-Strukturierung des Codes in C++-Strukturen, etc. und Veröffentlichung von Ergebnissen.
- Wegen der geplanten Veränderungen an den Schwerpunkten der in der Abteilung betriebenen Entwicklungen und fehlender Hardware wurde in Juli 2012 nach der Diskussion bisheriger Ergebnisse vorgeschlagen, das Projekt vorzeitig abzuschließen.
- Die eventuell noch zu leistenden Arbeiten an konservativen Advektionsverfahren wurden gestrichen. Das Vorhaben soll mit einem Abschlussbericht im März 2013 enden, der u.a. die Möglichkeiten weiterer Entwicklungen bezüglich Casulli-Algorithmen und Hardware-Beschleunigung erörtern soll.

2 Mathematisches Verfahren

2.1 2DV-Flachwassergleichungen

Zwecks Referenz werden hiermit die grundlegenden zweidimensionalen, vertikal gemittelten (2DV) Flachwassergleichungen in der transienten, nicht-konservativen Form für das nicht-rotierende kartesische Bezugssystem (x, y) angegeben:

$$\begin{aligned} H(u_t + uu_x + vu_y) &= -gH\eta_x + (\nu H u_x)_x + (\nu H u_y)_y + \gamma_T u_a - \gamma u \\ H(v_t + uv_x + vv_y) &= -gH\eta_y + (\nu H v_x)_x + (\nu H v_y)_y + \gamma_T v_a - \gamma v \\ \eta_t + (Hu)_x + (Hv)_y &= 0 \end{aligned} \tag{1}$$

Sie bestehen aus zwei Impulsgleichungen und der Kontinuitätsgleichung für die vertikal gemittelten horizontalen Geschwindigkeitskomponenten $u(x, y, t)$, $v(x, y, t)$ in der x- und y-Richtung, und den Wasserstand (Position der freien Oberfläche) $\eta(x, y, t)$; t ist die Zeit. Die Gesamtwassertiefe ist durch $H(x, y, t) = \max(0, \eta(x, y, t) + h(x, y))$ gegeben, wobei $h(x, y)$ die Wassertiefe (Bathymetrie) ist, man vergleiche Abb. 3. Die vorgegebene horizontale (Wirbel-)Viskosität wird mit ν bezeichnet. Die Reibungsterme sind wie folgt hergeleitet: $\gamma_B u + \gamma_T(u - u_a) = (\gamma_B + \gamma_T)u - \gamma_T u_a = \gamma u - \gamma_T u_a$, wobei u_a die Windgeschwindigkeit ist – und stellen die kombinierte Reibung an der Oberfläche und an dem Boden dar. Die Erdbeschleunigung ist mit g bezeichnet.

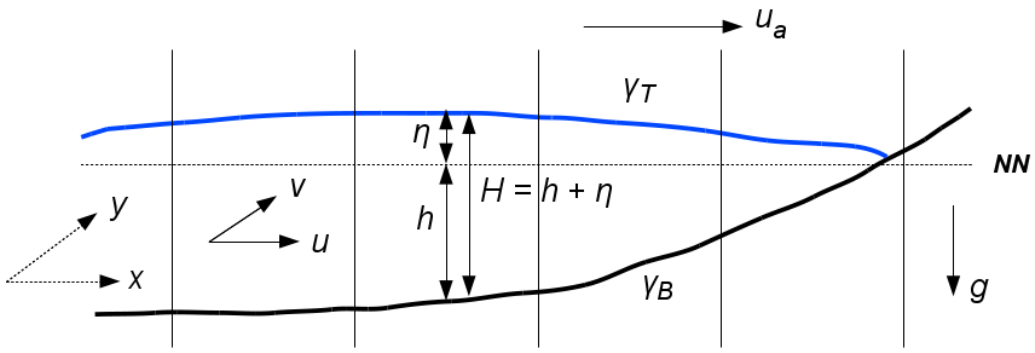

Abbildung 3: *Flachwassergleichungen: Erklärung der Variablen.*

2.2 Das allgemeine semi-implizite Finite-Differenzen Schema

Der hier betrachtete Lösungsalgorithmus basiert auf der allgemeinen Finite-Differenzen (FD) Methode für Flachwassergleichungen, wie von Casulli et al. [9, 12, 21] formuliert. Eine Analyse von den grundlegenden Gleichungen (1) erlaubt eine präzise Bestimmung dieser Terme, die auf implizite bzw. semi-implizite Weise diskretisiert werden müssen und von diesen Termen, die als unkritisch für die Stabilität explizit betrachtet werden können [9].

Auf diese Weise kann ein bemerkenswert stabiles, genaues und rechnerisch effizientes Schema formuliert werden, das ohne Verluste bezüglich der Robustheit und Einfachheit auch auf drei Dimensionen erweitert werden kann [12, 13, 21], eine nicht-hydrostatische Version inbegriffen [10]. Als Konsequenz ist das vertikal integrierte Schema (2DV, bzw. $2Dxy$) ein Grenzfall eines Standard-3D-Schemas, falls nur eine Netzschicht genommen wird. Die resultierenden Implementationen – der Code TRIM-2D und nachfolgend TRIM-3D – wurden in mehreren ingenieurpraktischen Projekten und wissenschaftlichen Untersuchungen angewendet, auch besonders intensiv in den neunziger Jahren in der BAW, z.B. [12, 21, 29, 28].

In diesem Beitrag wird das oben genannte semi-implizite Finite-Differenzen Verfahren angewendet, ergänzt mit einer neuen wichtigen Erweiterung für eine verbesserte, massenerhaltende nicht-lineare Betrachtung von Überflutung und Trockenfallen (*wetting and drying*), beispielsweise für Watt-Flächen, Vorländer eines Flusses, etc. Eine detaillierte Analyse [11, 16] dieser nicht-linearen Methode ist ohne Begrenzung der Allgemeinheit für eine Weiterentwicklung des gegebenen Schemas für orthogonale unstrukturierte Netze vorhanden (Code UNTRIM) [15, 17, 18, 40, 52]. Zur Vereinfachung, ist die im weiteren Text verwendete Notation ist identisch mit den in diesem Kapitel zitierten Referenzen.

2.3 Das diskrete nicht-lineare 2DV-Schema

Das numerische Finite-Differenzen Schema wird auf einem viereckigen FD-Netz formuliert, das aus $N_x \times N_y$ rechteckigen Polygonen von der Länge Δx und Breite Δy besteht (N_x und N_y sind die Anzahl der Netz-Polygone in x- bzw. y-Richtung). Das auf diese Weise definierte horizontale Rechengebiet Ω ist fixiert und soll den vom Fluid besetzten (und zeitlich veränderlichen) Bereich $\Omega(t) = \{(x, y) \in \Omega : H(x, y, t) > 0\}$ während der Simulationszeit $t \geq 0$ vollständig beinhalten. Das Schema ist transient (instationär) mit einem diskreten Zeitschritt Δt; die Zeitschritte werden mit n nummeriert.

Die diskretisierten Variablen werden in versetzten (gestaffelten) Positionen im FD-Netz definiert, was durch die spezifische Nummerierung widergespiegelt wird. Die Netz-Polygone (hier Rechtecke) sind mit Indizes (i, j) nummeriert, die gleichzeitig die Position in der Polygon-Mitte bezeichnen. Auf diese Weise wird die Wassertiefe (Bathymetrie) $h_{i,j}$, der Wasserstand $\eta^n_{i,j}$ und die Gesamtwassertiefe $H^n_{i,j}$ im Zeitschritt n in den Polygon-Mitten definiert.

Die diskretisierten Geschwindigkeitskomponenten u und v werden dagegen auf den Polygon-Kanten vorgegeben. In den diskretisierten Grundgleichungen wird dadurch eine Nummerierung gewählt, die die versetzte Position der Geschwindigkeitspunkte betont: u wird mit halber Ganzzahl i und voller Ganzzahl j gekennzeichnet, dementsprechend wird v mit voller Ganzzahl i und halber Ganzzahl j vorgegeben. Hiermit bekommen wir für alle vier Polygon-Kanten die Notation $u^n_{i\pm\frac{1}{2},j}$ und $v^n_{i,j\pm\frac{1}{2}}$. Passend dazu, falls die Bathymetrie und die Gesamtwassertiefe auf der Position der Polygon-Kanten verwendet wird, wird dies auch durch halbe Indizes betont, z.B. $H^n_{i\pm\frac{1}{2},j}$ and $H^n_{i,j\pm\frac{1}{2}}$. Die Reibungsfaktoren γ_T, γ und die Viskosität ν sind auch separat für x- und y-Richtung auf den Polygon-Kanten vorgegeben.

Im Prinzip kann das Schema die Darstellung von teilweise benetzten (d.h. teilweise überfluteten und teilweise trockenen) Netz-Polygonen erlauben, ohne die Notwendigkeit der Bestimmung einer präzisen Verteilung von überfluteten Bereichen. Man redet in diesem Fall von der Berücksichtigung einer SubGrid-Geometrie [16, 45]. Hierdurch kann die Gesamtwassertiefe H auf die folgende nicht-lineare Weise definiert werden:

$$H(x,y,\eta^n) = \int_{-\infty}^{\eta^n} p(x,y,z)dz = \max\left(0, h(x,y) + \eta^n\right) \tag{2}$$

Es wird eine Hilfsfunktion $p(x,y,z)$ (auch *Porosität* genannt) eingeführt, um die Verteilung der Gesamttiefe in einem Netz-Polygon zu repräsentieren:

$$p(x,y,z) = \begin{cases} 1 & \text{falls} \quad h(x,y) + z > 0 \\ 0 & \text{falls} \quad h(x,y) + z \leq 0 \end{cases} \tag{3}$$

Die Netz-Polygone können formell beschrieben werden als $\Omega_{i,j}(\eta^n) = \{(x,y) : x_{i-\frac{1}{2}} \leq x \leq x_{i+\frac{1}{2}}, y_{j-\frac{1}{2}} \leq y \leq y_{j+\frac{1}{2}}$, wo $H(x,y,\eta^n) > 0\}$. Deswegen kann der überflutete Bereich von einem Netz-Polygon $p_{i,j}$, das dem mittleren Wasserstand über dem gegebenen Polygon $\eta^n_{i,j}$ entspricht, mit der Anwendung der Hilfsfunktion (3) durch ein horizontales Integral ausgerückt werden:

$$0 \leq p_{i,j}(\eta^n_{i,j}) = \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} p(x,y,\eta^n_{i,j})dxdy \leq \Delta x \Delta y \tag{4}$$

In der Folge kann die Gesamtwassertiefe entlang einer Kante entsprechend einer vorgegeben SubGrid-Verteilung variieren. Der Querschnittsbereich von der Wassersäule über dem benetzten (überfluteten) Anteil einer Polygon-Kante – d.h. die vom Wasser durchströmte Querfläche – kann exemplarisch für die u-Kante wie folgt ausgedrückt werden:

$$A^n_{i+\frac{1}{2},j} = \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} H(x_{i+\frac{1}{2}}, y, \eta^n_{i+\frac{1}{2},j})dy \tag{5}$$

Hiermit kann man auch die durschnittliche Gesamtwassertiefe entlang einer teilweise überfluteten Polygon-Kante definieren:

$$H^n_{i+\frac{1}{2},j} = \frac{A^n_{i+\frac{1}{2},j}}{\int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} p(x_{i+\frac{1}{2}}, y, \eta^n_{i,j})dy} = \frac{A^n_{i+\frac{1}{2},j}}{\Delta y^n_{i+\frac{1}{2},j}} \tag{6}$$

wobei die benetzte Kantenlänge $\Delta y^n_{i+\frac{1}{2},j}$ in der Zeit variieren kann (man merke den Unterschied zum konstanten Abstand zwischen den Netz-Knoten Δy). Das Wasservolumen in jedem Polygon kann dadurch auf ähnliche Weise ausgedrückt werden:

$$0 \leq V_{i,j}(\eta^n_{i,j}) = \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} H(x,y,\eta^n_{i,j})dxdy = \int_{-\infty}^{\eta^n_{i,j}} p_{i,j}(z)dz \tag{7}$$

Mit so definierter Geometrie von Überflutung und Trockenfallen können die Impulsgleichungen in Flachwassergleichungen (1) auf semi-implizite Weise diskretisiert werden [12]. Hierbei wird die θ-Methode für die Gradienten der freien Oberfläche verwendet, d.h. sie werden mit dem Faktor θ zwischen den Zeitebenen $n+1$ und n gewichtet:

$$\begin{aligned}\left(H^n_{i+\frac{1}{2},j} + \Delta t \gamma^n_{i+\frac{1}{2},j}\right) u^{n+1}_{i+\frac{1}{2},j} &= \mathbf{F} u^n_{i+\frac{1}{2},j} \\ -g\frac{\Delta t}{\Delta x} H^n_{i+\frac{1}{2},j} &\left[\theta\left(\eta^{n+1}_{i+1,j} - \eta^{n+1}_{i,j}\right) + (1-\theta)\left(\eta^n_{i+1,j} - \eta^n_{i,j}\right)\right] + \Delta t \gamma_T u_a\end{aligned} \tag{8}$$

$$\begin{aligned}\left(H^n_{i,j+\frac{1}{2}} + \Delta t \gamma^n_{i,j+\frac{1}{2}}\right) v^{n+1}_{i,j+\frac{1}{2}} &= \mathbf{F} v^n_{i,j+\frac{1}{2}} \\ -g\frac{\Delta t}{\Delta y} H^n_{i,j+\frac{1}{2}} &\left[\theta\left(\eta^{n+1}_{i,j+1} - \eta^{n+1}_{i,j}\right) + (1-\theta)\left(\eta^n_{i,j+1} - \eta^n_{i,j}\right)\right] + \Delta t \gamma_T v_a\end{aligned} \tag{9}$$

wo $\mathbf{F}u = Hu^* + \Delta t[(\nu u_x)_x + (\nu u_y)_y]^*$ ein expliziter, nicht-linearer Finite-Differenzen Operator ist, der die (explizite) Diskretisierung von advektiven und viskosen Termen beinhaltet (Kap. 2.5.1). Der Term γ_T betrifft die Windreibung mit einer vorgeschriebenen Windgeschwindigkeit (u_a, v_a) und der Term $\gamma = r_B\sqrt{u_*^2 + v_*^2}$ repräsentiert die Reibung nach der Anwendung des o.g. genannten Advektion-Diffusion Operator auf die Geschwindigkeit. r_B ist der Reibungskoeffizient, der für vertikal gemittelte Flachwassergleichungen auf vielfältige Weise definiert werden kann, beispielsweise nach Chézy ist $r_B = g/C_z^2$.

Aufgrund der Tatsache, dass $\eta_t = H_t$, erhält man durch die Integration der Kontinuitätsgleichung (1) auf der Polygon-Fläche den folgenden Ausdruck:

$$\left[\int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}}\int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} H_{i,j}\,dxdy\right]_t + \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}}\left[(Hu)_{i+\frac{1}{2}} - (Hu)_{i-\frac{1}{2}}\right]dy + \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}}\left[(Hv)_{j+\frac{1}{2}} - (Hv)_{j-\frac{1}{2}}\right]dx = 0 \tag{10}$$

Mit der Bemerkung, dass das erste Integral in (10) das Wasservolumen darstellt und die gemittelte Geschwindigkeitskomponente u entlang der y-Kante konstant bleibt (genauso, wie v entlang der x-Kante), erhält man eine Finite-Volumen Form der Kontinuitätsgleichung, die eine präzise Bilanzierung des Volumens über dem Netz-Polygon (i,j) darstellt:

$$\begin{aligned}V_{i,j}(\eta^{n+1}_{i,j}) = V_{i,j}(\eta^n_{i,j}) &- \theta\Delta t\left[A^n_{i+\frac{1}{2},j}u^{n+1}_{i+\frac{1}{2},j} - A^n_{i-\frac{1}{2},j}u^{n+1}_{i-\frac{1}{2},j} + A^n_{i,j+\frac{1}{2},j}v^{n+1}_{i,j+\frac{1}{2}} - A^n_{i,j-\frac{1}{2}}v^{n+1}_{i,j-\frac{1}{2}}\right] \\ &- (1-\theta)\,\Delta t\left[A^n_{i+\frac{1}{2},j}u^n_{i+\frac{1}{2},j} - A^n_{i-\frac{1}{2},j}u^n_{i-\frac{1}{2},j} + A^n_{i,j+\frac{1}{2},j}v^n_{i,j+\frac{1}{2}} - A^n_{i,j-\frac{1}{2}}v^n_{i,j-\frac{1}{2}}\right]\end{aligned} \tag{11}$$

Die Geschwindigkeit wurde dabei genauso, wie für den Wasserstand, mit der θ-Methode zwischen den Zeitebenen gewichtet [12].

Ein Gleichungssystem für die Wasserstände $\eta^{n+1}_{i,j}$ entsteht durch Einführung von Impulsgleichungen (8-9) unter Berücksichtigung von (6) in die Kontinuitätsgleichung (11):

$$
\begin{aligned}
V_{i,j}(\eta_{i,j}^{n+1}) - \theta^2 g \frac{\Delta t^2}{\Delta x} \left[\left(\frac{\Delta y H^2}{H + \Delta t \gamma} \right)^n_{i+\frac{1}{2},j} \left(\eta_{i+1,j}^{n+1} - \eta_{i,j}^{n+1} \right) - \left(\frac{\Delta y H^2}{H + \Delta t \gamma} \right)^n_{i-\frac{1}{2},j} \left(\eta_{i,j}^{n+1} - \eta_{i-1,j}^{n+1} \right) \right] \\
-\theta^2 g \frac{\Delta t^2}{\Delta y} \left[\left(\frac{\Delta x H^2}{H + \Delta t \gamma} \right)^n_{i,j+\frac{1}{2}} \left(\eta_{i,j+1}^{n+1} - \eta_{i,j}^{n+1} \right) - \left(\frac{\Delta x H^2}{H + \Delta t \gamma} \right)^n_{i,j-\frac{1}{2}} \left(\eta_{i,j}^{n+1} - \eta_{i,j-1}^{n+1} \right) \right] \\
= b_{i,j}^n
\end{aligned}
\tag{12}
$$

Der Vektor $b_{i,j}^n$ auf der rechten Seite des Systems beinhaltet alle (bereits) bekannten Terme in der Gleichung (12) aus der Zeitebene n. Zwecks Vereinfachung der mathematischen Formeln werden Hilfsterme eingeführt: $G_x = \mathbf{F}u + \Delta t \gamma_T u_a$ und $G_y = \mathbf{F}v + \Delta t \gamma_T v_a$. Mit dieser Notation bekommt man für $b_{i,j}^n$:

$$
\begin{aligned}
b_{i,j}^n &= V_{i,j}(\eta_{i,j}^n) \\
&\quad -\theta \Delta t \left[\left(\frac{\Delta y H G_x}{H + \Delta t \gamma} \right)^n_{i+\frac{1}{2},j} - \left(\frac{\Delta y H G_x}{H + \Delta t \gamma} \right)^n_{i-\frac{1}{2},j} + \left(\frac{\Delta x H G_y}{H + \Delta t \gamma} \right)^n_{i,j+\frac{1}{2}} - \left(\frac{\Delta x H G_y}{H + \Delta t \gamma} \right)^n_{i,j-\frac{1}{2}} \right] \\
&\quad - (1 - \theta) \Delta t \left[(\Delta y H u)^n_{i+\frac{1}{2},j} - (\Delta y H u)^n_{i-\frac{1}{2},j} + (\Delta x H v)^n_{i,j+\frac{1}{2}} - (\Delta x H v)^n_{i,j-\frac{1}{2}} \right]
\end{aligned}
\tag{13}
$$

Der nicht-lineare Charakter von der Gleichung (12) entsteht durch die Definition von dem Volumen $V_{i,j}(\eta_{i,j}^n)$, das ein Integral von der stückweise konstanten, aber diskontinuierlichen Hilfsfunktion p ist. Mit der Einführung einer neuen Variable $\zeta_{i,j} = \eta_{i,j}^{n+1}$ kann das Gleichungssystem (12) in einer kompakten Form ausgedrückt werden:

$$
\mathbf{V}(\zeta) + \mathbf{T}\zeta = \mathbf{b} \tag{14}
$$

Der Vektor $\mathbf{V}$, von der Länge $N_x N_y$, beinhaltet die Zellen-Volumen. Die Struktur der entstehenden Wasserstandsmatrix $\mathbf{T}$ (von der Große $N_x N_y \times N_x N_y$) kann mit der Gleichung (12) gefunden werden. Die Hilfsvariablen $R = \Delta x H^2/(H + \Delta t \gamma)$ und $S = \Delta y H^2/(H + \Delta t \gamma)$ einführend, bekommt man für die Elemente der Matrix $\mathbf{T}$:

$$
\begin{aligned}
t_{i,j} &= \theta^2 g \frac{\Delta t^2}{\Delta x} \left(S_{i+\frac{1}{2},j} + S_{i-\frac{1}{2},j} \right) + \theta^2 g \frac{\Delta t^2}{\Delta y} \left(R_{i,j+\frac{1}{2}} + R_{i,j-\frac{1}{2}} \right), \\
t_{i+\frac{1}{2},j} &= -\theta^2 g \frac{\Delta t^2}{\Delta x} S_{i+\frac{1}{2},j}, \qquad t_{i-\frac{1}{2},j} = -\theta^2 g \frac{\Delta t^2}{\Delta x} S_{i-\frac{1}{2},j} \\
t_{i,j+\frac{1}{2}} &= -\theta^2 g \frac{\Delta t^2}{\Delta y} R_{i,j+\frac{1}{2}}, \qquad t_{i,j-\frac{1}{2}} = -\theta^2 g \frac{\Delta t^2}{\Delta y} R_{i,j-\frac{1}{2}}
\end{aligned}
\tag{15}
$$

Die Wasserstandsmatrix $\mathbf{T}$ ist eindeutig penta-diagonal. Die Terme auf der Hauptdiagonalen sind von der Summe von Beiträgen von allen umgebenden Kanten eines Netz-Polygons (i, j) gebildet. Die Werte, die aus den benachbarten Zellen stammen, sind auf den seitlichen Diagonalen zu finden. So strukturiert, erfüllt diese Matrix die Bedingungen für die Konvergenz

des Newton-Verfahrens zur Lösung von nichtlinearen Gleichungsystemen. Nach dem Umschreiben von (14) in die sog. kanonische Form für das Newton-Verfahren bekommt man:

$$\mathbf{f} = \mathbf{V}(\zeta) + \mathbf{T}\zeta - \mathbf{b} = 0 \tag{16}$$

Das Problem kann man weiterhin in die iterative Form des Newton-Verfahrens umwandeln. Mit m als Iterationsindex kann man sie wie folgt formulieren:

$$\zeta^{m+1} = \zeta^m + \Delta\zeta = \zeta^m - [\mathbf{f}'(\zeta^m)]^{-1}\mathbf{f}(\zeta^m) = \zeta^m - \left[\mathbf{V}'(\zeta^m) + \mathbf{T}\right]^{-1}\left[\mathbf{V}(\zeta^m) + \mathbf{T}\zeta^m - \mathbf{b}\right] \tag{17}$$

Nach der Definition (7) ist die Ableitung des Zell-Volumens $\mathbf{V}'(\zeta) = p_{i,j}(\zeta_{i,j}^m)$ eine Single-Diagonalmatrix mit nicht-negativen Werten, die zu $\mathbf{T}$ in (17) addiert wird. In der Tat, wegen (7) haben wir auf der Hauptdiagonalen von $\mathbf{V}'$ Werte $0 < p_{i,j} < \Delta x\Delta y$ für teilweise gefüllte, $p_{i,j} = \Delta x\Delta y$ für vollständig gefüllte und Nullwerte $p_{i,j} = 0$ für trocken gefallene Zellen. Im Ausnahmefall, nämlich falls ein Polygon (i, j) vollständig trocken und von genauso trocken gefallenen Polygonen umgeben ist, muss ein Polygon von der Systemmatrix eliminiert werden (oder der entsprechende Term in der Hauptdiagonale wird zu 1 gesetzt).

Jede Iteration des Newton-Verfahrens (17) kann mit der Methode der Konjugierten Gradienten durchgeführt werden, um $\Delta\zeta$ vom System

$$[\mathbf{V}'(\zeta^m) + \mathbf{T}]\Delta\zeta = \mathbf{f}(\zeta^m) \tag{18}$$

zu bestimmen, wobei eine Pre-Konditionierung dieses Systems bei einer größeren Anzahl der Polygone notwendig werden kann.

Die Iterationen im Newton-Verfahren werden durchgeführt, bis eine vorgegebene Genauigkeit ε im Sinne der Vektornorm $|\Delta\zeta| \leq \varepsilon$ erreicht wird. Aufgrund der milden Nicht-Linearität des Systems ist die erforderliche Anzahl von Iterationen sehr moderat, und zwar mit nur einer Iteration falls sich die Konfiguration von trocken gefallenen und überfluteten Zellen in einem Zeitschritt nicht ändert. Die mathematisch formelle Diskussion dieser mild nicht-linearen Methode steht für den allgemeineren Fall von unstrukturierten Netzen zur Verfügung [11].

Nach der Bestimmung neuer Wasserstände $\eta_{i,j}^{n+1}$ können die beiden Geschwindigkeitskomponenten mit Hilfe der Impulsgleichungen (8-9) gefunden werden. Die neuen Gesamtwassertiefen pro Kante werden mit Hilfe von (6) bestimmt.

Der so resultierende Algorithmus findet seine Anwendung für den Fall der orthogonalen unstrukturierten Netze im Code UnTRIM^2 [16, 45], für den Fall der kartesischen Quadtrees im neuen Verfahren nach Stelling [61] und für traditionelle FD-Netze (d.h. genauso wie hier dargestellt) im SubGrid/Multigrid-Verfahren, das in einem laufenden Projekt *Effizienz- und Genauigkeitssteigerung der Modellierung der Hydrodynamik der Flüsse mit einem kombinierten Multigrid- und SubGrid-Ansatz* (A39530270002) für numerische Untersuchungen verwendet wird.

2.4 Das diskrete 2DV-Schema ohne Subgrids

Eine Version von der Flachwassergleichung (12), welche die SubGrid-Geometrie nicht berücksichtigt, kann sehr einfach hergeleitet werden, indem man, anstelle der zeitlich variablen benetzten (nassen) Kantenlängen, konstante Werte verwendet, $\Delta y^n_{i\pm\frac{1}{2},j} = \Delta y$, $\Delta x^n_{i,j\pm\frac{1}{2}} = \Delta x$. Die Bathymetrie bzw. Wassertiefe auf der Fläche eines Polygons wird dann konstant und für die Position in der Mitte eines Polygons wie folgt bestimmt:

$$h_{i,j} = \max\left(h_{i+\frac{1}{2},j}, h_{i-\frac{1}{2},j}, h_{i,j+\frac{1}{2}}, h_{i,j-\frac{1}{2}}\right) \tag{19}$$

Die Gesamtwassertiefen pro Kante werden mit Hilfe von der folgenden Beziehung gefunden:

$$H^n_{i+\frac{1}{2},j} = h_{i+\frac{1}{2},j} + \max(\eta^n_{i,j}, \eta^n_{i+1,j}) \qquad H^n_{i,j+\frac{1}{2}} = h_{i,j+\frac{1}{2}} + \max(\eta^n_{i,j}, \eta^n_{i,j+1}) \tag{20}$$

Angenommen, dass wir nur die vollständig überfluteten Polygone betrachten, können die beiden Seiten der Gleichungen (12) durch die gesamte Zellfläche $\Delta x \Delta y$ dividiert werden:

$$\begin{aligned}
\eta^{n+1}_{i,j} &- \theta^2 g \frac{\Delta t^2}{\Delta x^2}\left[\left(\frac{H^2}{H+\gamma\Delta t}\right)^n_{i+\frac{1}{2},j}\left(\eta^{n+1}_{i+1,j} - \eta^{n+1}_{i,j}\right) - \left(\frac{H^2}{H+\gamma\Delta t}\right)^n_{i-\frac{1}{2},j}\left(\eta^{n+1}_{i,j} - \eta^{n+1}_{i-1,j}\right)\right] \\
&- \theta^2 g \frac{\Delta t^2}{\Delta y^2}\left[\left(\frac{H^2}{H+\gamma\Delta t}\right)^n_{i,j+\frac{1}{2}}\left(\eta^{n+1}_{i,j+1} - \eta^{n+1}_{i,j}\right) - \left(\frac{H^2}{H+\gamma\Delta t}\right)^n_{i,j-\frac{1}{2}}\left(\eta^{n+1}_{i,j} - \eta^{n+1}_{i,j-1}\right)\right] \\
&= b^n_{i,j}
\end{aligned} \tag{21}$$

wobei die rechte Seite des Systems $b^n_{i,j}$ wie folgt geschrieben werden kann:

$$\begin{aligned}
b^n_{i,j} = \eta^n_{i,j} &- \theta\frac{\Delta t}{\Delta x}\left[\left(\frac{HG_x}{H+\gamma\Delta t}\right)^n_{i+\frac{1}{2},j} - \left(\frac{HG_x}{H+\gamma\Delta t}\right)^n_{i-\frac{1}{2},j}\right] \\
&- \theta\frac{\Delta t}{\Delta y}\left[\left(\frac{HG_y}{H+\gamma\Delta t}\right)^n_{i,j+\frac{1}{2}} - \left(\frac{HG_y}{H+\gamma\Delta t}\right)^n_{i,j-\frac{1}{2}}\right] \\
&- (1-\theta)\frac{\Delta t}{\Delta x}\left[(Hu)^n_{i+\frac{1}{2},j} - (Hu)^n_{i-\frac{1}{2},j}\right] - (1-\theta)\frac{\Delta t}{\Delta y}\left[(Hv)^n_{i,j+\frac{1}{2}} - (Hv)^n_{i,j-\frac{1}{2}}\right]
\end{aligned} \tag{22}$$

Definiert auf einem Finite-Differenzen-Netz von $N_x \times N_y$ Polygonen, bilden die Gleichungen (21) ein lineares System mit $N_x N_y$ Gleichungen mit einer penta-diagonalen Matrix, das gelöst werden muss, um die Wasserstände $\eta^{n+1}_{i,j}$ zu bestimmen. Aufgrund der Tatsache, dass die Wassertiefen $H^{n+1}_{i\pm\frac{1}{2},j}$, $H^{n+1}_{i,j\pm\frac{1}{2}}$ immer nicht-negativ sind, ist die Matrix dieses Systems positiv definiert und symmetrisch. Ein solches Gleichungssystem kann auch ohne Anwendung des nichtlinearen iterativen Newton-Schemas mit der Methode der Konjugierten Gradienten gelöst werden. Komplett trocken gefallene Zellen müssen entdeckt und aus dem Gleichungssystem entfernt werden. Jedoch selbst wenn die Polygon-Kanten als komplett nass

oder vollständig trocken behandelt werden, verliert das im vorigen Abschnitt 2.3 beschriebene nicht-lineare Schema seine Allgemeinheit nicht und öffnet die Möglichkeit der Verfolgung einer veränderlichen Uferlinie um mehr als eine (ganze) Zelle während eines einzigen Zeitschrittes.

Genauso wie im vorigen Abschnitt 2.3 können nach der Bestimmung neuer Wasserstände $\eta_{i,j}^{n+1}$ die beiden Geschwindigkeitskomponenten mit Hilfe der Impulsgleichungen (8-9) gefunden werden.

2.5 Diskretisierung advektiver und viskoser Terme

2.5.1 Traditionelle Ansätze

Die advektiven und viskosen Terme sind im Verfahren nach Casulli auf explizite Weise diskretisiert. Im Grenzfall, falls die Vernachlässigung der Advektion und der horizontalen Terme zulässig ist, reduziert sich der explizite Operator $\mathbf{F}$ in (8) und (9) in 2DV-Fall zu $\mathbf{F}u_{i+\frac{1}{2},j}^n = H_{i+\frac{1}{2},j}^n u_{i+\frac{1}{2},j}^n$.

Traditionell wird eine von der Courant-Zahl unabhängige Euler-Lagrange-Diskretisierung für die Advektion verwendet (gemeint ist Charakteristikenverfahren) und die horizontalen viskosen Terme $(1/H)\nabla \cdot (H\nu\nabla u)$ werden in der vereinfachten Form $\nabla \cdot (\nu\nabla u))$ dargestellt. In diesem Fall kann man (exemplarisch für die u-Geschwindigkeitskomponente) den expliziten Operator wie folgt ausdrücken:

$$\mathbf{F}u_{i+\frac{1}{2},j}^n = (\mathbf{F}_{adv} + \mathbf{F}_{visc})\, u_{i+\frac{1}{2},j}^n = H_{i+\frac{1}{2},j}^n u_{i+\frac{1}{2},j}^* + \Delta t\, [(\nu u_x)_x + (\nu u_y)_y]_{i+\frac{1}{2},j}^* \tag{23}$$

Als $u_{i+\frac{1}{2},j}^*$ wird die Geschwindigkeit auf der Zeitebene t^n gekennzeichnet, die am Fußpunkt der lagrangschen Trajektorie von den umgebenden Gridpunkten interpoliert wird. Die Trajektorie wird durch Integration des Geschwindigkeitsfelds von dem Geschwindigkeitspunkt $(i+\frac{1}{2}, j)$ anfangend rückwärts in der Zeit von der Ebene t^{n+1} zu t^n bestimmt. Die Methodik dieses Verfahrens ist im Detail von Casulli et al. [9, 13, 21] für lineare und von Deussfeld [28] für die quadratische Interpolation beschrieben und wird an dieser Stelle nicht näher erörtert. Im entwickelten Code wird die Integration entlang der Trajektorie basierend auf der lokalen CFL-Zahl mit der Euler-Methode zweiter Ordnung realisiert, mit linearer Interpolation des Geschwindigkeitsfelds.

Der Ausdruck $\mathbf{F}_{visc}u_{i+\frac{1}{2},j}^n = [(\nu u_x)_x(\nu u_y)_y]_{i+\frac{1}{2},j}^*$ ist die Diskretisierung des horizontalen viskosen Terms auch am Fuß der lagrangschen Trajektorie, es werden zentrale Differenzen verwendet:

$$\begin{aligned}\mathbf{F}_{visc}u^n_{i+\frac{1}{2},j} = &+ \frac{\nu\Delta t}{\Delta x^2}\left(u^n_{i-\frac{1}{2},j} - 2u^n_{i+\frac{1}{2},j} + u^n_{i+\frac{3}{2},j}\right)\\ &+ \frac{\nu\Delta t}{\Delta y^2}\left(u^n_{i+\frac{1}{2},j-1} - 2u^n_{i+\frac{1}{2},j} + u^n_{i+\frac{1}{2},j+1}\right)\end{aligned} \tag{24}$$

$$\begin{aligned}\mathbf{F}_{visc}v^n_{i,j+\frac{1}{2}} = &+ \frac{\nu\Delta t}{\Delta y^2}\left(v^n_{i,j-\frac{1}{2}} - 2v^n_{i,j+\frac{1}{2}} + v^n_{i,j+\frac{3}{2}}\right)\\ &+ \frac{\nu\Delta t}{\Delta x^2}\left(v^n_{i-1,j+\frac{1}{2}} - 2v^n_{i,j+\frac{1}{2}} + v^n_{i+1,j+\frac{1}{2}}\right)\end{aligned} \tag{25}$$

Falls anstatt des Charakteristikenverfahrens für die Advektion die klassische *upwind*-Methode (und dann mit einer Zeitschritt-Beschränkung durch die Courant-Zahl) für die Advektion angewendet wird [9], ist der Operator $\mathbf{F}$ die Summe von der diesmal lokal in der direkten Umgebung eines Polygons berechneten advektiven $\mathbf{F}_{adv}$ und der viskosen Beiträge. Es wird das folgende Upwind-Schema mit einem konstanten oder variablen Zeitschritt $\Delta t < \Delta x/|u|_{max}$ gewählt:

$$\begin{aligned}A_m &= \frac{1}{2}\left(\frac{1}{2}(u^n_{i+\frac{3}{2},j} + u^n_{i+\frac{1}{2},j}) - \frac{1}{2}|u^n_{i+\frac{3}{2},j} + u^n_{i+\frac{1}{2},j}|\right)\\ A_p &= \frac{1}{2}\left(\frac{1}{2}(u^n_{i-\frac{1}{2},j} + u^n_{i+\frac{1}{2},j}) + \frac{1}{2}|u^n_{i-\frac{1}{2},j} + u^n_{i+\frac{1}{2},j}|\right)\\ B_m &= \frac{1}{2}\left(\frac{1}{2}(v^n_{i,j+\frac{3}{2}} + v^n_{i,j+\frac{1}{2}}) - \frac{1}{2}|v^n_{i,j+\frac{3}{2},j} + v^n_{i,j+\frac{1}{2}}|\right)\\ B_p &= \frac{1}{2}\left(\frac{1}{2}(v^n_{i,j-\frac{1}{2}} + u^n_{i,j+\frac{1}{2}}) + \frac{1}{2}|v^n_{i,j-\frac{1}{2}} + v^n_{i,j+\frac{1}{2}}|\right)\end{aligned} \tag{26}$$

$$\begin{aligned}\mathbf{F}_{adv}u^n_{i+\frac{1}{2},j} = u^n_{i+\frac{1}{2},j} &- \frac{\Delta t}{\Delta x}\left[A_m(u^n_{i+\frac{3}{2},j} - u^n_{i+\frac{1}{2},j}) + A_p(u^n_{i+\frac{1}{2},j} - u^n_{i-\frac{1}{2},j})\right]\\ &- \frac{\Delta t}{\Delta y}\left[B_m(u^n_{i+\frac{1}{2},j+1} - u^n_{i+\frac{1}{2},j}) + B_p(u^n_{i+\frac{1}{2},j} - u^n_{i+\frac{1}{2},j-1})\right]\\ \mathbf{F}_{adv}v^n_{i,j+\frac{1}{2}} = v^n_{i,j+\frac{1}{2}} &- \frac{\Delta t}{\Delta x}\left[A_m(v^n_{i+1,j+\frac{1}{2}} - v^n_{i,j+\frac{1}{2}}) + A_p(v^n_{i,j+\frac{1}{2}} - v^n_{i-1,j+\frac{1}{2}})\right]\\ &- \frac{\Delta t}{\Delta y}\left[B_m(v^n_{i,j+\frac{3}{2}} - v^n_{i,j+\frac{1}{2}}) + B_p(v^n_{i,j+\frac{1}{2}} - v^n_{i,j-\frac{1}{2}})\right]\end{aligned} \tag{27}$$

Für die Zwecke der in diesem Dokument beschriebenen Untersuchungen, die primär nur performance-orientiert sind, wurde in allen weiteren Berechnungen für die Advektion das Upwind-Schema (27) mit einem konstanten Zeitschritt gewählt.

Es ist zu beachten, dass es zahlreiche Möglichkeiten für die Festlegung der expliziten Operatoren für die Advektion $\mathbf{F}_{adv}$ und Diffusion $\mathbf{F}_{visc}$ gibt. Im Fall der Advektion werden sie im nächsten Kapitel 2.5.2 näher behandelt.

2.5.2 Alternative Advektionsverfahren

In der Klasse der *upwind*-Verfahren für allgemeine numerische Schemata für natürliche Strömungen mit freier Oberfläche sind jene Advektionsverfahren besonders attraktiv, die für alle Abflussarten und bei der Anwesenheit von dynamisch verlaufenden Prozessen von Trockenfallen und Überfluten universell verwendet werden können. Da die Erhaltung von Masse, Impuls und Energie gleichzeitig in einem nicht-linearen Advektionsverfahren schwierig ist, konstruiert man für Flachwasserströmungen Schemata, die ihre Eigenschaften abhängig von der Abflussart (dynamisch) verändern. Diskussion dieser Verfahren für gestaffelte Netze ist von Duinmeijer und Stelling [62] im strukturierten und von Kramer und Stelling im unstrukturierten Fall [44] gegeben. Verfahren dieser Art finden ihre Anwendung auch im D-FLOW-Modellierungssystem von Deltares (DELFT-3D) [27]. In diesem Vorhaben wurden ansatzweise die einfachsten Verfahren dieser Art (Genauigkeit erster Ordnung) seriell implementiert und rudimentär getestet. Die Konstruktion des Schemas basiert man auf drei prinzipiellen Eigenschaften, die man bei unterschiedlichen Abflussarten wahlweise betont:

1. Massenerhaltung – für Wassertiefen überall, insbesondere Erhaltung positiver Wassertiefen bei Überflutungen und Trockenfallen.

2. Impulserhaltung – angewendet im Fall einer Erweiterung in der Strömung, um Wechselsprünge (d.h. Übergänge Schießen/Strömen) richtig reproduzieren zu können.

3. Energieerhaltung – im Fall der Verengung in der Strömung, um z.B. die (beschleunigten) Übergänge Strömen/Schießen zu erfassen.

Beispielsweise bei einer Strömung über einem Wehr mit Übergangen in der Abflussart, ist die strikte Energieerhaltung wichtig bei dem Übergang zu schießender Strömung über der Wehrkrone (Verengung) und die Impulserhaltung bei dem danach stattfindenden Wechselsprung (mit der Dissipation der Energie). Die Massenerhaltung muss überall erfüllt werden.

Nach Duinmeijer und Stelling [62] können die nicht-linearen advektiven Terme durch die folgenden Approximationen erster Ordnung der dargestellt werden (hiermit beschränken wir uns auf die x-Richtung):

$$\left(u\frac{\partial u}{\partial x}\right)_{i+1/2} \approx \max(u_{\rightarrow},0)\left(\frac{u_{i+1/2}-u_{i-1/2}}{\Delta x}\right)+\min(u_{\leftarrow},0)\left(\frac{u_{i+3/2}-u_{i+1/2}}{\Delta x}\right) \tag{28}$$

wo die Ansätze für die advektiven Geschwindigkeiten $u_{\rightarrow}$ und $u_{\leftarrow}$ abhängig von den Erhaltungsgesetzen formuliert werden. Für die Impulserhaltung:

$$u_{\rightarrow}=\frac{\bar{q}_i}{\bar{H}_{i+1/2}}, \qquad u_{\leftarrow}=\frac{\bar{q}_{i+1}}{\bar{H}_{i+1/2}} \tag{29}$$

wobei $\bar{H}_{i+1/2}=\frac{h_i+h_{i+1}}{2}$, $\bar{q}_i=\frac{q_{i+1/2}+q_{i-1/2}}{2}$ und $q_{i+1/2}=H^*_{i+1/2}u_{i+1/2}$. Die Gesamtwassertiefe über einer Kante $H^*_{i+1/2}$ wird durch *Upwinding* erster Ordnung bestimmt:

$$H^*_{i+1/2} = \begin{cases} H_i, & \text{falls} \quad u_{i+1/2} > 0 \\ H_{i+1}, & \text{falls} \quad u_{i+1/2} < 0 \\ \max(\eta_i, \eta_{i+1}) + \min(h_i, hi + 1), & \text{falls} \quad u_{i+1/2} = 0 \end{cases} \tag{30}$$

Für die Energieerhaltung:

$$u_{\rightarrow} = \frac{1}{2}\left(u_{i+1/2} + u_{i-1/2}\right), \qquad u_{\leftarrow} = \frac{1}{2}\left(u_{i+1/2} + u_{i+3/2}\right) \tag{31}$$

Schließlich, für ein nicht-konservatives Schema, wird für die Geschwindigkeit der Advektion die Geschwindigkeit selbst gewählt:

$$u_{\rightarrow} = u_{\leftarrow} = u_{i+1/2} \tag{32}$$

Auf diese Weise liegt der Unterschied zwischen der Impulserhaltung, Energieerhaltung und keiner Erhaltung in der Art der Interpolation der Advektionsgeschwindigkeit aus dem Strömungsfeld. Als Kriterium, wie man zwischen der Impulserhaltung und Energieerhaltung wechselt, kann ein einfacher dynamischer Algorithmus verwendet werden: Falls $\frac{u_{i+1/2}-u_{i-1/2}}{\Delta x} > \varepsilon > 0$ benutze Gl. 31, sonst Gl. 29. Bei der Wahl des Koeffizienten ε kann so gesteuert werden, dass die Energieerhaltung nur bei starken Verengungen der Strömung verwendet wird, sonst die (übliche) Impulserhaltung. Für die Details wird auf die Veröffentlichung von Duinmeijer und Stelling [62] verwiesen.

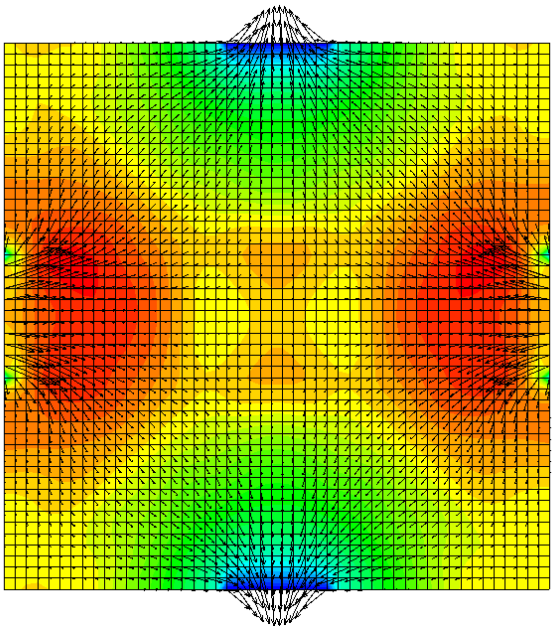

Abbildung 4: *Tests von Einlauf- und Auslauf-Randbedingungen auf einem Finite-Differenzen Netz.*

2.6 Anfangs- und Randbedingungen

Bei der Vorgabe der typischen Ein- und Auslauf-Randbedingungen, Abb. 4, wurde die einfachste Vorgehensweise gewählt, die auch in TRIM und UnTRIM praktiziert wird. Die Randbedingungen für die Wasserstände werden direkt nach der Lösung des nicht-linearen Gleichungssystems gesetzt und für die Geschwindigkeitskomponenten nach der finalen Bestimmung aus Gl. (8-9) vorgegeben. Die Reibung an den seitlichen Rändern wird durch die *Slip*-Randbedingung bei dem expliziten Diffusionsoperator (24) berücksichtigt.

Die Vorgabe der Anfangsbedingungen erfolgt durch Initialisierung der Werte für alle Wasserstände η und die beiden Geschwindigkeitskomponenten u und v.

3 Parallele Implementierung

3.1 Der zu implementierende Algorithmus

Der zu implementierende Algorithmus basiert auf der Version des 2DV-Schemas ohne Subgrids (Kap. 2.4) und kann wie folgt skizziert werden:

- Einrichtung des Netzes und der Geometrie
- Initialisierung und Speicherzuweisung
- Die Zeitschleife
  - Bestimmung der Gesamtwassertiefen $H^n_{i,j}$, $H^n_{i\pm\frac{1}{2},j}$, $H^n_{i,j\pm\frac{1}{2}}$, zusammen mit dem Benetzungszustand (Trockenfallen/Überfluten)
  - Feststellung des Zeitschrittes Δt für die Zeitebene n falls das *Upwind*-Verfahren verwendet wird
  - Explizite Advektion- und Diffusion-Operatoren $\mathbf{F}u^n_{i\pm\frac{1}{2},j}$, $\mathbf{F}v^n_{i,j\pm\frac{1}{2}}$
  - Assemblieren der linearen Anteile der Wasserstand-Matrix $\mathbf{T}$ und des Vektors der rechten Seite $\mathbf{b}$
  - Newton-Iterationen für die neuen Wasserstände: $\zeta^m = \eta^{n+1}_{i,j}$ bis $|\Delta\zeta| = |\zeta^m - \zeta^{m-1}| \leq \varepsilon$:
    - Berechnung mit ζ^m des Vektors auf der rechten Seite $\mathbf{f}(\zeta^m) = \mathbf{V}(\zeta^m)+\mathbf{T}\zeta^m-\mathbf{b}$
    - Bestimmen der nicht-linearen Anteile $\mathbf{V}'(\zeta^m)$ und die Addition von $\mathbf{V}'$ zu der Wasserstandsmatrix $\mathbf{T}$
    - Iterative Lösung des linearen Gleichungssystems $[\mathbf{V}'(\zeta^m) + \mathbf{T}]\Delta\zeta = \mathbf{f}(\zeta^m)$
  - Bestimmung neuer Geschwindigkeitskomponenten $u^{n+1}_{i\pm\frac{1}{2},j}$, $v^{n+1}_{i,j\pm\frac{1}{2}}$
  - Optionale Ausgabe der Ergebnisse

Der prinzipielle semi-implizite Algorithmus ist kompakt genug, um schnell unter Verwendung einer beliebig geeigneten Programmiersprache oder mathematischer Software programmiert zu werden. Mit einer Ausnahme für die neue nicht-lineare Betrachtung des Überflutens und des Trockenwerdens und den verwendeten Advektionsverfahren ist dieser Algorithmus mit dem Rechenkern des TRIM-2D-Code [13, 21] sehr ähnlich.

3.2 Gewählter Ansatz

3.2.1 Verwendung von CUDA

Das Hauptziel dieser Untersuchung ist u.a. die Wirtschaftlichkeit der Hardware-orientierten Programmierung zu beurteilen, insbesondere unter Verwendung neuer Entwicklungswerkzeuge. Angesichts der sehr versprechenden Resultate der GPU-Programmierung gibt es eine

laufende Diskussion, wie die bereits existierenden, größeren und komplexen *legacy* Codes zu für heterogene Hardware zu adaptieren sind, da es bisher keine allgemeingültigen Empfehlungen gibt. Beispielsweise variieren die bisherigen Ansätze zwischen der Adaptierung der rechnerisch intensivsten Anteile [30], Verwendung von semi-automatischen Skripten (um bereits mit OPENMP parallelisierten Code-Abschnitte in die auf GPU lauffähige Kernel umzuwandeln [24]), bis zu kompletten Re-Implementierungen von einfachen (expliziten) Verfahren [25] bis zu gesamten größeren Solvern [22].

Es existieren bereits serielle, vektorisierte und unter Verwendung von OPENMP parallelisierte Implementierungen des im früheren Absatz 3.1 dargestellten Algorithmus, nämlich in der Form der Rechenkerne von TRIM-2D und TRIM-3D [13, 21, 29]. Trotzdem, wie bereits in Abs. 1.4.3 erwähnt, ist der in diesem Vorhaben gewählte Ansatz, einen neuen Code vom Anfang an unter Verwendung der früheren Versionen als Referenz, zu schreiben. Ein Grund dafür ist, dass diese *legacy* FORTRAN Codes (ca. 1990-99) die nicht-lineare Behandlung von Überfluten und Trockenfallen nicht beinhalten. Sie wurde erst für das Nachfolge-Programm UNTRIM ab 2009 für unstrukturierte Netze eingeführt [11, 16]. Ein weiterer Grund ist, dass Vorstudien mit Elementen von TRIM-3D mit den OPENMP-ähnlichen *accelerator directives* und mit CUDA FORTRAN [55] im Jahr 2010 gezeigt haben (Kap. 1.4.3), dass die Verwendung einer neuen, GPU-orientierten Programmiersprache, wie CUDA C [48] oder OPENCL [41] deutlich vorteilhafter wäre, nennenswerte Geschwindigkeitssteigerungen unter Verwendung von GPUs zu erreichen. Diese Entscheidung wurde zusätzlich durch positive Erfahrungen der wissenschaftlichen Gemeinschaft mit diesen Sprachen und die Verfügbarkeit von numerischen Bibliotheken, die die GPU-optimierte Vektor-parallele Operationen implementieren, beeinflusst.

3.2.2 GPU-optimierte numerische Bibliotheken

In diesem Projekt wird ein Programm von Anfang an mit CUDA C geschrieben, wobei eine gleichzeitig entstehende serielle Version für die Referenz im Sinne der Korrektheit in allen Phasen der Entwicklung verwendet wurde. Soweit wie möglich wurden die derzeitig verfügbaren Versionen der Bibliotheken THRUST [34] und CUSP [1] genutzt. THRUST ist eine Bibliothek von Algorithmen aus dem Bereich der linearen Algebra, die Hardwarebeschleunigte Operationen auf Datenvektoren implementiert, mit einem Interface, das der *C++ Standard Template Library* ähnlich ist. CUSP ist eine Bibliothek von generischen parallelen Algorithmen für Berechnungen mit Matrizen und Graphen. Beide Bibliotheken wurden mit CUDA C mit C++-Erweiterungen geschrieben und liefern hoch optimierte CUDA-Kernel für arithmetisch intensive Operationen auf GPUs. Es muss angemerkt werden, dass Codes, die THRUST- und CUSP-Bibliotheken verwenden, auch auf serielle Hosts ohne Veränderung lauffähig sind.

Beide Bibliotheken wurden im Laufe der Implementierung des Casulli-Algorithmus für eine konsistente Speicherverwaltung auf dem Gerät (GPU) und dem Host (CPU) verwendet, genauso wie für die Übertragung von einzelnen Schleifen des Verfahrens in die GUU-spezifischen CUDA-Kernel, sowie für die Adaptierung des gesamten Konjugierten-

Gradienten Gleichungslösers. Nur für komplexere, nicht einfach mit vektoriellem Ansatz von THRUST und CUSP adaptierbare Anteile des Codes, wie z.B. die Advektion-Schemata, wurden CUDA-Kernel geschrieben, zusammen mit ihren seriellen Äquivalenten. Die Verwendung der spezialisierten Bibliotheken hat den Vorteil, dass sie das Aufrechterhalten einer deutlich Vektor-orientierten Code-Struktur vereinfachen und dass sie in andere als die Hardware-spezifische CUDA-C-Sprache übertragen werden können – nur durch den Austausch von Kerneln für grundlegende Operationen.

3.2.3 Der resultierende Code

Das Netz, Bathymetrie und Wasserstände werden auf dem Host (CPU) initialisiert und danach zum Device (GPU) transferiert, wobei die geometrischen Daten, die auf dem Host residieren, werden später für Interpolationen im versetzten Netz benutzt (zwecks Ausgabe für knoten-orientierte Post-Prozessoren, wie z.B. Tecplot). Obwohl generell durch die CPU kontrolliert, wird die gesamte Zeitschleife ausschließlich auf dem Gerät (GPU) ausgeführt, was entscheidend für die Performance des Verfahrens ist.

Eine Ausnahme in der Zeitschleife bildet die Ausgabe der Ergebnisse in Dateien. Um ausgegeben zu werden, müssen die Resultate zurück von dem GPU-Speicher zum Hauptspeicher (unter Kontrolle der CPU) transferiert werden. Das Problem dieses Performance-raubenden Speichertransfers von der GPU zur CPU kann derzeitig z.B. mit der Verwendung der CUDA *streaming*-Technologie gelöst werden: Diese Technologie erlaubt gleichzeitige Berechnungen und Transfers bzw. gleichzeitige Operationen auf CPU und GPU. Diese I/O-Problematik wird in dieser Studie, die sich auf den Rechenkern konzentriert, nicht näher behandelt. Dieses Problem von Transfer-Latenzzeiten kann vielleicht in der Zukunft z.B. durch die erscheinenden APUs (*Accelerated Processing Units*) deutlich gemildert werden, da sie CPU und GPU auf einem Chip verschmelzen sollen. Sie sollten auch die Notwendigkeit des Managements von getrennten Speichern des Hosts und Devices entfernen. Ein anderer Grund ist, dass die THRUST und CUSP Bibliotheken derzeitig mit der *streaming*-Technologie nicht kompatibel sind. Wie bereits erwähnt, von zentralenr Interesse beim derzeitigen Entwicklungsstand der Hardware ist, dass sowohl für die GPUs, APUs sowie SIMD-Prozessoren der aktuellen CPUs der Vektorisierungsgrad des Codes für die Leistung entscheidend ist.

Aufgrund der Anwendung der Vektor-orientierten THRUST- und CUSP-Bibliotheken und der Einführung von kompakten CUDA-Kerneln, ist die Struktur des Codes durchaus anders im Vergleich zum seriellen Referenz-Code: Praktisch jede Schleife über Datenvektoren wurde verändert, es gibt separate Speicherzuweisungen für die Host- und Device-Speicher und den expliziten Datenübertragungen zwischen Device und Host. Der resultierende Code ähnelt gewissermaßen den alten *legacy*-Codes für Vektorrechner oder den Codes, die intensiv mathematische Bibliotheken bzw. spezielle Routinen verwenden, die korrekte Vektorisierung des Codes garantieren, wie z.B. den von TELEMAC [33].

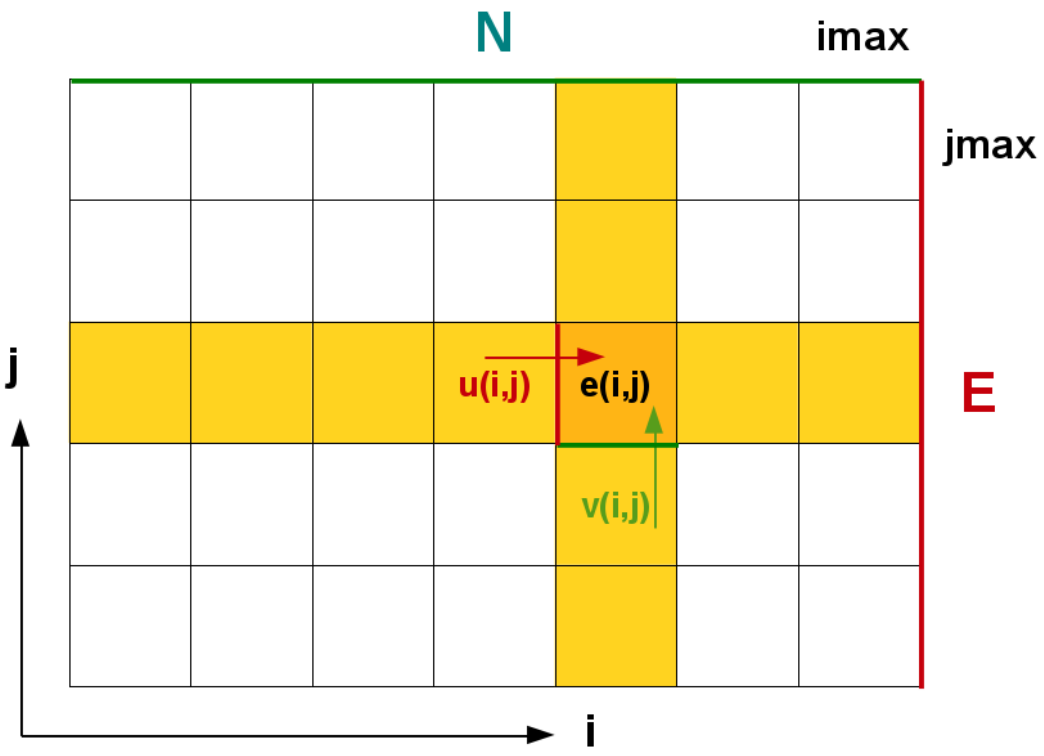

Abbildung 5: *Das Finite-Differenzen Netz.*

3.3 Auserwählte Details der Implementierung

In diesem Kapitel werden illustrativ einige auserwählte Details der Implementation auf vereinfachte Weise dargestellt.

Es wird ein zweidimensionales viereckiges Finite-Differenzen Netz verwendet, Abb. 5, das den gesamten Simulationsbereich abdeckt und nach den Himmelsrichtungen orientiert ist. In der Berechnung werden alle Netz-Polygone betrachtet, ohne Eliminierung aufgrund des Trockenfallens. Die Reihenfolge der Indizes im Netz wird so gewählt, dass die u- und v-Komponente der Geschwindigkeit auf den unteren (S) und linken (E) Polygon-Kanten die gleichen Indizes wie die Variablen in der Mitte dieses Polygons haben, wie z.B. Wasserstände – man vergleiche die Abb. 5. Das führt dazu, dass die Werte auf dem östlichen und nördlichen Rand des FD-Netzes in getrennten Datenfeldern gespeichert werden, hauptsächlich für die Vorgabe der Randbedingungen.

Es ist natürlich für zweidimensionale FD-Netze, dass die diskretisierten Variablen unter Verwendung von doppelten Indizen `(i,j)` organisiert werden. Wegen der vektoriellen Natur des Codes werden in diesem Fall Einzel-Indizes `ij` benutzt, im Zusammenhang zu den doppelten Indizen in der Form `ij=j*imax+i`. Man vergleiche die resultierenden FD-Schablonen (*Stencils*) in der Abb. 6. Mit solcher vektorieller Index-Struktur werden die Zugriffe in einem FD-*Stencil* mit einem konstanten Schritt im Speicher erfolgen, was besonders

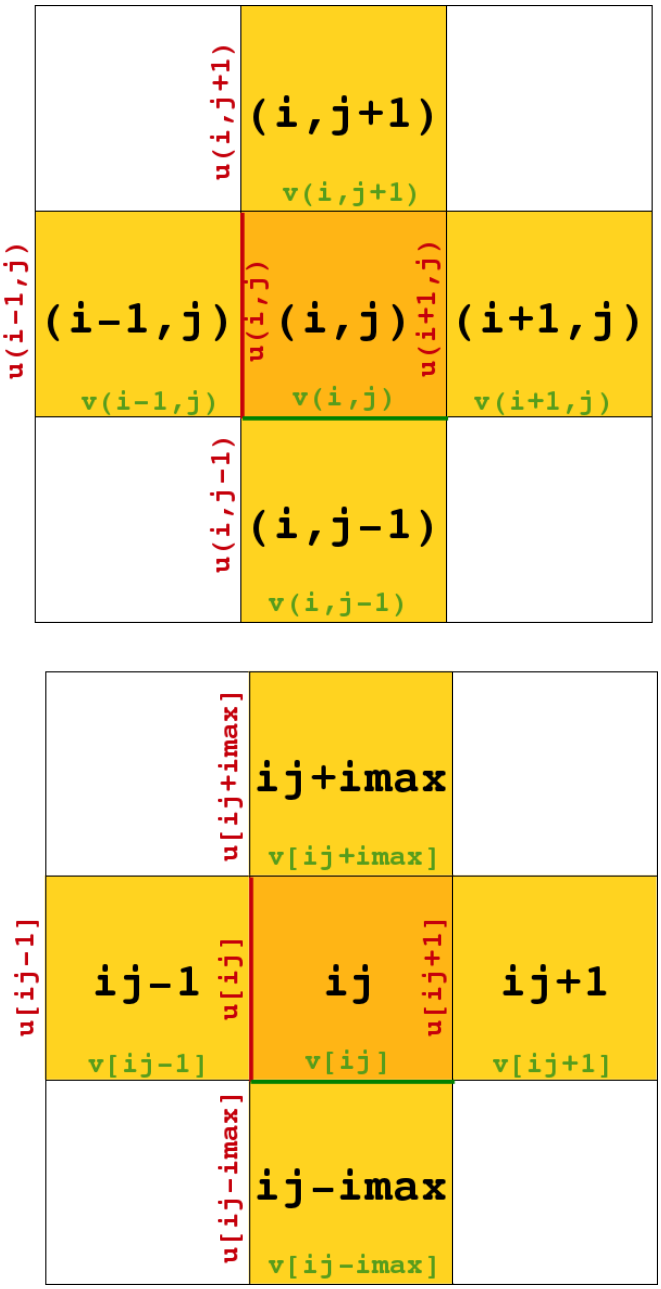

Abbildung 6: *Doppelte und einzelne Indizes im FD-Netz.*

vorteilhaft für die Optimierung des Codes ist. Der besondere Vorteil ist, dass der CUDA-Compiler die Speicher-Hierarchie einer GPU nicht explizit, sondern als Cache nutzen kann.

Der Code wurde gleichzeitig für den Tandem CPU+GPU und für eine serielle Ausführung auf einer CPU geschrieben. Die Übersetzung einer entsprechenden parallelen oder seriellen Version wird durch Compiler-Direktiven gesteuert. Es werden allgemeine Typen für Ganz- und Gleitkommazahlen verwendet (`indx` bzw. `real`), was ein einfaches Wechseln zwischen der Genauigkeit erlaubt. Durch die Anwendung von THRUST- und CUSP-Bibliotheken unterscheidet sich der Code im Vergleich zu einem reinen CUDA C/C++-Code insbesondere durch die Vorgehensweise mit der Speicherzuweisung beim Host- und Device-Speicher und mit den Transfers dazwischen. Sie werden mit den Bibliotheken-Routinen realisiert, die sowohl für CPU, wie für GPU gelten. Die meisten Variablen werden als CUSP-Objekte `array1d`, also Datenvektoren, deklariert. Sie können dadurch als solche in den Bibliotheken-Routinen verwendet werden, wie Kopieren, Transformieren, Reduktion, etc. Sie können auch mit den Methoden des Objekts selbst bearbeitet werden, z.B. (Re-)Allozieren des Datenfeldes für die Wasserstände mit `eta.resize(ncells)`. Falls diese Objekte als einfache C-Datenvektoren außerhalb der Bibliotheken z.B. als Parameter der C-Routinen verwendet werden müssen, kann auf die Datenfelder dieser Objekte durch eine Zeiger-Zuweisung von THRUST, `thrust::raw_pointer_cast` zugegriffen werden.

Hiermit Code-Fragmente aus diversen Routinen, die o.g. Ausführungen illustrieren.

```
#ifdef ongpu
  typedef cusp::device_memory MemorySpace;
#else
  typedef cusp::host_memory MemorySpace;
#endif
[...]
  typedef cusp::array1d<indx, MemorySpace> IndxArray;
  typedef cusp::array1d<real, MemorySpace> RealArray;
  typedef cusp::array1d<indx, cusp::host_memory> HostIndxArray;
  typedef cusp::array1d<real, cusp::host_memory> HostRealArray;
[...]
  RealArray eta; real* eta_ptr;
  HostRealArray hb_host;
  RealArray hb; real* hb_ptr;
  RealArray v;   real* v_ptr;
[...]
  eta.resize(ncells); eta_ptr = thrust::raw_pointer_cast(&eta[0]);
  hb_host.resize(ncells,real(0));
  hb.resize(ncells);  hb_ptr = thrust::raw_pointer_cast(&hb[0]);
  u.resize(ncells,ini.uini); u_ptr = thrust::raw_pointer_cast(&u[0]);
[...]
  // bathymetry per cell transferred to the device: hb_host -> hb
  thrust::copy (hb_host.begin(), hb_host.end(), hb.begin());
[...]
  // htx[ij] = hx[ij] + eta[ij]
  thrust::transform(hx.begin(), hx.end(),
```

```
                    eta.begin(), htx.begin(),
                    sur_plus_bot<real>(real(0)));
[...]
  template<typename T>
    struct sur_plus_bot : public thrust::binary_function<T,T,T>
    {
      const T level;
      sur_plus_bot(T _level) : level(_level) {}
      __host__ __device__
      T operator()(const T& sur, const T& bot) const {
        return max(level,sur+bot); // level can be 0 or some epsilon
      }
    }; // end sur_plus_bot
```

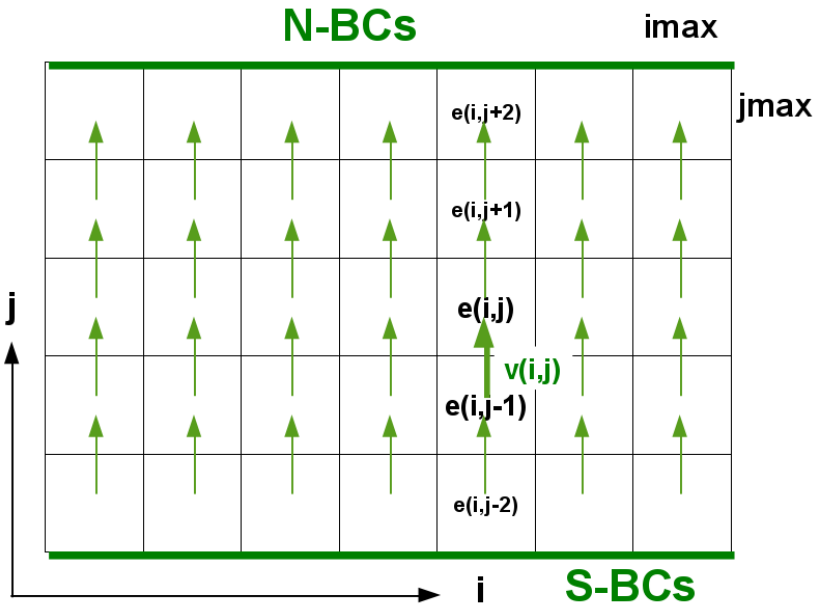

Abbildung 7: *Berechnung der finalen v-Gechwindikeitskomponente im gegebenen FD-Netz.*

Des Weiteren werden drei Implementierungen der Routine zur finalen Berechnung der v-Geschwindigkeitskomponenten nach der Formel (9) aus den Gradienten der Wasserstände und mit der Vorgabe der Randbedingungen diskutiert. Die Vorgehensweise wird schematisch auf der Abb. 7 im FD-Netz dargestellt. Die zu berechnende Formel ist:

$$v_{i,j+\frac{1}{2}}^{n+1} = \frac{H_{i,j+\frac{1}{2}}^{n}\left[\mathbf{F}v_{i,j+\frac{1}{2}}^{n} - g\frac{\Delta t}{\Delta y}\theta\left(\eta_{i,j+1}^{n+1} - \eta_{i,j}^{n+1}\right) - g\frac{\Delta t}{\Delta y}\left(1-\theta\right)\left(\eta_{i,j+1}^{n} - \eta_{i,j}^{n}\right)\right]}{\left(H_{i,j+\frac{1}{2}}^{n} + \Delta t\gamma_{i,j+\frac{1}{2}}^{n}\right)} \tag{33}$$

Zwecks einfacherer Präsentation wird die eigentliche Berechnung leicht vereinfacht, es werden z.B. keine spezifischen Randbedingungen gesetzt (nur exemplarisch die Dirichletschen). Die serielle Version des Codes (man bemerke die Anweisung `__host__`) sieht wie folgt aus:

```
extern "C"
__host__ void
new_v_host( real* v, real* vnorth,
            const real* fv, const real* eta, const real* etan,
            const real* hty, const real* gamv,
            const indx imax, const indx jmax, real g,
            const real levele, const real levelu,
            const real dt, const real dy, const real thetae,
            const real velbcy0, const real velbcy1) {

  indx i,j,ij;

  for (j=1; j<jmax; j++)
    for (i=0; i<imax; i++) {
      ij=imax*j+i;
      if (hty[ij] > levelu) {
v[ij] = (fv[ij] - thetae*g*dt/dy*(eta[ij]-eta[ij-imax])
              - (1.0-thetae)*g*dt/dy*(etan[ij]-etan[ij-imax]))
      * hty[ij]/max(hty[ij]+dt*gamv[ij],levele);
      } else {
        v[ij] = real(0);
    }
  }
  // BCs simplified
  for (i=0; i<imax; i++) {
    v[i]=velbcy0;
    vnorth[i]=velbcy1;
  }
} // new_v_host
```

Diese Routine kann als ein CUDA-Kernel (Anweisung `__global__`) für die Ausführung auf einer GPU umgeschrieben werden. Die doppelten Indizes in `for`-Schleifen haben jetzt eine zweifache Bedeutung, sie ordnen die Thread-Prozessoren – organisiert zweidimensional mit Indizes `i,j` und zusätzlich abstrahierend von der Hardware in *blocks* und *threads* – dem Index `ij` im Datenvektor.

Alternativ kann man anstatt der `for`-Schleifen eine Lösung mit `if` anwenden (im Code auffälig auskommentiert). Wegen der asynchronen Ausführung in Multiprozessoren der GPU, muss (falls notwendig) eine Synchronisierung von *threads* mittels `__syncthreads()` erfolgen – sonst wäre die Reproduzierbarkeit der Ergebnisse nicht gewährleistet.

```
extern "C"
__global__ void
new_v_devi (real* v, real* vnorth,
            const real* fv, const real* eta, const real* etan,
            const real* hty, const real* gamv,
            const indx imax, const indx jmax, const real g,
            const real levele, const real levelu,
            const real dt, const real dy, const real thetae,
            const real velbcy0, const real velbcy1) {

  indx i,j,ij,istart,jstart,istep,jstep;

  istart = blockIdx.x * blockDim.x + threadIdx.x;
  jstart = blockIdx.y * blockDim.y + threadIdx.y;
  istep  = blockDim.x * gridDim.x;
  jstep  = blockDim.y * gridDim.y;

  //i = blockIdx.x * blockDim.x + threadIdx.x;
  //j = blockIdx.y * blockDim.y + threadIdx.y;
  // if (i<imax && (j>0 && j<jmax)) {

  for (j=jstart+1; j<jmax; j=j+jstep)
    for (i=istart; i<imax; i=i+istep) {
      ij=j*imax+i;
      if (hty[ij] > levelu) {
      v[ij] = (fv[ij] - thetae*g*dt/dy*(eta[ij]-eta[ij-imax])
              - (1.0-thetae)*g*dt/dy*(etan[ij]-etan[ij-imax]))
      * hty[ij]/max(hty[ij]+dt*gamv[ij],levele);
      }
      else {
        v[ij] = real(0);
      }
    }
  // BCs
  for (i=istart; i<imax; i=i+istep) {
    v[i]=velbcy0;
    vnorth[i]=velbcy1;
  }
  __syncthreads();
} // new_v_device
```

Die Entscheidung, welche von den beiden oben dargestellten Routinen genommen wird, wird bei der Übersetzung durch die Compiler-Direktiven vorgegeben.

Beim Aufruf eines CUDA-Kernels auf der GPU muss bei einem CUDA Programm eine Dimensionierung des Netzes (*grid*) von Blocks von Threads erfolgen, die der Hardware der GPU-Karte entsprechen muss (Abb. 8). Hiermit ein Beispiel, das adäquat für eine Nvidia GeForce GTX480 und das hier behandelte Programm ist.

```
    dim3 dimBlock( min(16,imax), min(16,jmax));
    dim3 dimGrid ( min(255, (imax+dimBlock.x-1)/dimBlock.x),
                   min(255, (jmax+dimBlock.y-1)/dimBlock.y));
[...]
#ifdef ongpu
        new_v_devi <<<dimGrid,dimBlock>>>
          (v_ptr, vnorth_ptr, fv_ptr,
           eta_ptr, etan_ptr, hty_ptr, gamv_ptr,
           imax, jmax, g, simparams.epsh, dt, dy,
           simparams.thetae, bcs.velbcy0, bcs.velbcy1);
#else
        new_v_host
          (v_ptr, vnorth_ptr, fv_ptr,
           eta_ptr, etan_ptr, hty_ptr, gamv_ptr,
           imax, jmax, g, simparams.epsh, dt, dy,
           simparams.thetae, bcs.velbcy0, bcs.velbcy1);
#endif
```

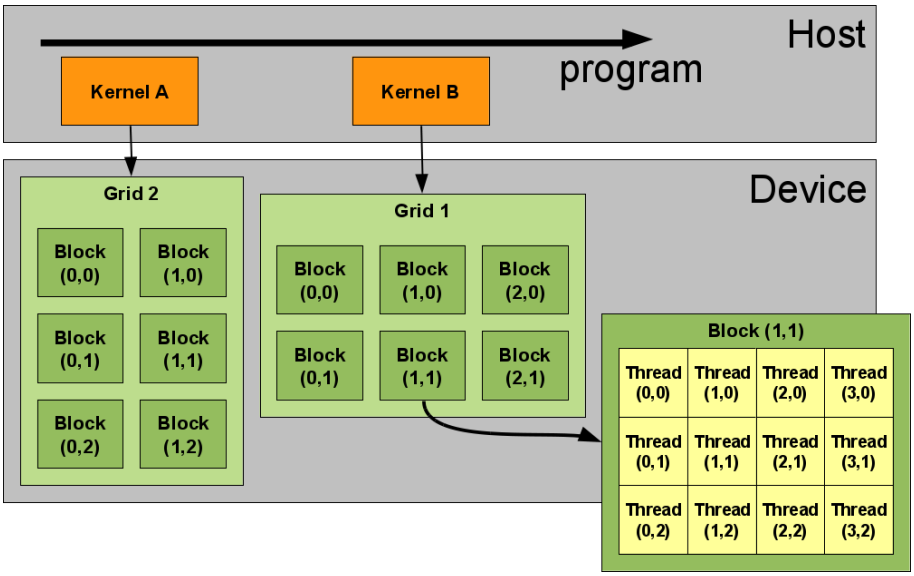

Abbildung 8: *Ausführung eines CUDA-Programms: Die CPU steuert die Abläufe und Aufrufe der Kernel-Funktionen, die mit unterschiedlichen Konfigurationen grids–blocks–threads auf der GPU ausgeführt werden.*

Anstatt einen CUDA-Kernel zu schreiben, kann man auch die Berechnung der v-Geschwindigkeitskomponente nach der Formel (9) vollständig vektoriell mit den THRUST-Funktionen realisieren. THRUST stellt viele fertige Vektor-Operationen bereit. Falls man eine komplexere Formel mit mehreren Vektoren, wie (9), verwenden möchte, soll eine abstrakte Template-Funktion erstellt werden. Sie wird danach für die Operation auf einem Vektor-Tupel (d.h. eine optimal für die Ausführung organisierte Gruppe von Datenvektoren) verwendet. Für

die gegebene Berechnung – hier wird $\theta = 1$ für die Vereinfachung gesetzt – sieht eine solche Template-Funktion wie folgt aus:

```
template <typename T>
  struct final_velocity_functor
  { T alpha;
    T beta;
    T gamma;

    final_velocity_functor ( T _alpha, T _beta, T _gamma)
      : alpha(_alpha), beta(_beta), gamma(_gamma) {}

    template <typename Tuple>
    __host__ __device__
        void operator()(Tuple t)
        {
          // computing final velocity without theta approach
          if (thrust::get<2>(t) > T(0)) {
             thrust::get<4>(t) =
               ( thrust::get<0>(t) - alpha * thrust::get<1>(t) )
                 * thrust::get<2>(t) /
                  (thrust::get<2>(t) + beta*thrust::get<3>(t) + gamma);
          }
          else {
            thrust::get<4>(t)=T(0);
          }
        }
  }; // final_velocity_functor
```

Die Implementation mittels THRUST beinhaltet zuerst die Berechnung von Gradienten der Wasserstände in einen Arbeitsfeld auf der GPU. Danach erfolgt die Berechnung einer Schleife mit einer Gruppe von Vektoren (Tupel) organisiert durch einen *Zip*-Operator (zusammenführen der Operanden) zur optimalen Ausführung auf einer GPU. Schließlich werden die Dirichletschen Randbedingungen gesetzt. Der Unterschied zu einem üblichen CUDA-Programm besteht im Wesentlichen darauf, dass man sich in diesem Fall um die Details der Konfiguration für die Ausführung auf der GPU (*grids–blocks–threads*) nicht kümmern muss.

```
 // compute eta[ij]-eta[ij-imax]
 thrust::fill(work_devi.begin(), work_devi.begin()+imax, real(0));
 thrust::transform(eta.begin()+imax, eta.end(), eta.begin(),
                   work_devi.begin()+imax, thrust::minus<real>());

 // the machine goes over all ij for computing v
 thrust::for_each(thrust::make_zip_iterator(thrust::make_tuple(
           fv.begin()+imax, work_devi.begin()+imax, hty.begin()+imax,
```

```
                gamv.begin()+imax, v.begin()+imax)),
            thrust::make_zip_iterator(thrust::make_tuple(
        fv.end(), work_devi.end(), hty.end(), gamv.end(), v.end())),
        final_velocity_functor<real>(g*dt/dy,dt,simparams.epsh));

// southern and northern Dirichlet BCs
thrust::fill(v.begin(),v.begin()+imax,bcs.velbcy0);
thrust::fill(vnorth.begin(),vnorth.end(),bcs.velbcy1);
```

Bei der Programmierung des 2DV-Codes wurde eine gemischte Vorgehensweise gewählt, mit CUDA-Kerneln für komplexe Berechnungen und THRUST-Methoden für einfachere Vektoroperationen. CUSP wird neben der Speicherzuweisungen und -transfers vor allem zur Lösung des linearen Gleichungssystems 21 verwendet.

4 Ergebnisse der Implementierung des 2DV-Verfahrens

4.1 Verifizierung

Die Verifizierung und Validierung des gegebenen Algorithmus in seinen verschiedenen Implementierungen wurde bereits mit zahlreichen Beispielen durchgeführt und veröffentlicht [13, 16, 19, 21, 29]. Der Vollständigkeit halber und um die nicht-lineare Betrachtung von Übefluten und Trockenfallen zu testen, wird ein klar nicht-linearer Testfall vorgestellt, für den eine analytische Lösung von Thacker [63] existiert. Das Beispiel betrifft eine periodische Bewegung eines kreisförmigen Wasserkörpers über einem Becken mit einem idealisierten, axialsymmetrisch paraboloidalen Boden, Abb. 9. Das anfängliche Profil der freien Oberfläche ist auch paraboloidal, aufwärts ausgebucht; die Anfangsgeschwindigkeit wird auf Null gesetzt. Unter dem Einfluss der Schwerkraft entwickelt sich eine periodische Bewegung mit einer Periode von $T = 2\pi/\omega$, wo ω die Frequenz ist.

Die Position der kreisförmigen Küstenlinie bewegt sich auch axialsymmetrisch hin und her um die natürliche Ruhe-Position. Falls man als Referenz die Ruhe-Position des Systems (flache horizontale Oberfläche, Abb. 9) nimmt, mit h_0 als die Wassertiefe in der Mitte des Beckens, L als den Radius der Gleichgewichtsküstenlinie und η_0 als anfängliche Verschiebung der ausgebuchten freien Oberfläche in der Mitte des Wasserkörpers, können die Bewegungsgleichungen wie folgt abgeleitet werden [63]:

$$\begin{aligned} \eta(x,y,t) &= h_0 \left\{ \frac{\sqrt{1-A^2}}{1-A\cos\omega t} - 1 - \frac{x^2+y^2}{L^2}\left[\frac{1-A^2}{(1-A\cos\omega)^2} - 1\right]\right\} \\ u(x,y,t) &= \frac{1}{1-A\cos\omega t}\left(\frac{1}{2}\omega x A \sin\omega t\right) \\ v(x,y,t) &= \frac{1}{1-A\cos\omega t}\left(\frac{1}{2}\omega y A \sin\omega t\right) \end{aligned} \tag{34}$$

wobei die Frequenz der Bewegung ω und der Parameter A wie folgt definiert sind:

$$\omega = \frac{\sqrt{8gh_0}}{L}, \qquad A = \frac{(h_0+\eta_0)^2 - h_0^2}{(h_0+\eta_0)^2 + h_0^2} \tag{35}$$

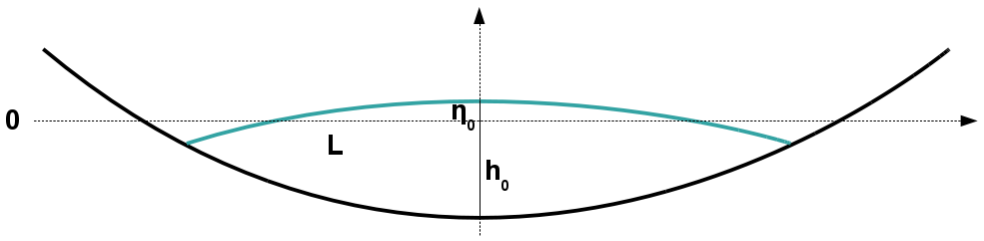

Abbildung 9: *Erklärungen für den Testfall nach Thacker.*

Für den gegebenen Testfall werden die geometrischen Parameter $L = 4.0$, $h_0 = 0.2$m, $\eta_0 = 0.01$m gewählt. Der quadratische Berechnungsbereich hat Abmessungen von 10m $\times$ 10m mit beiden x- und y-Koordinaten zwischen -5m and +5m variierend. Mit der Auflösung von 2000×2000 bekommt man Kanten-Längen der Netz-Polygonen $\Delta x = \Delta y = 0.005$m. Für diesen Parametersatz ist die resultierende Periode $T = 6.34$s; die Berechnung wird mit einem Zeitschritt von $\Delta t = 0.005$s für 50s, d.h. für über sieben Perioden der Bewegung durchgeführt. Der Fall ist nichtviskos, $\nu = 0$ und ohne Reibung, $\gamma = \gamma_T = 0$. Die Genauigkeit des (Konjugierte-Gradienten) Gleichungslösers ist zu 10^{-10} gesetzt, und für Newton-Iterationen 10^{-12}, was eine moderate Anzahl der inneren und bis zu drei äußeren Iterationen (wegen Nass/Trockenwerden) mit sich bringt. Die Bewegung der freien Oberfläche η ist in Abb. 10 in der Form von Zeitserien für zwei dezentrale und eine zentrale Position in der Mitte des Beckens dargestellt, wo auch die analytische Lösung als Referenz geplottet wird.

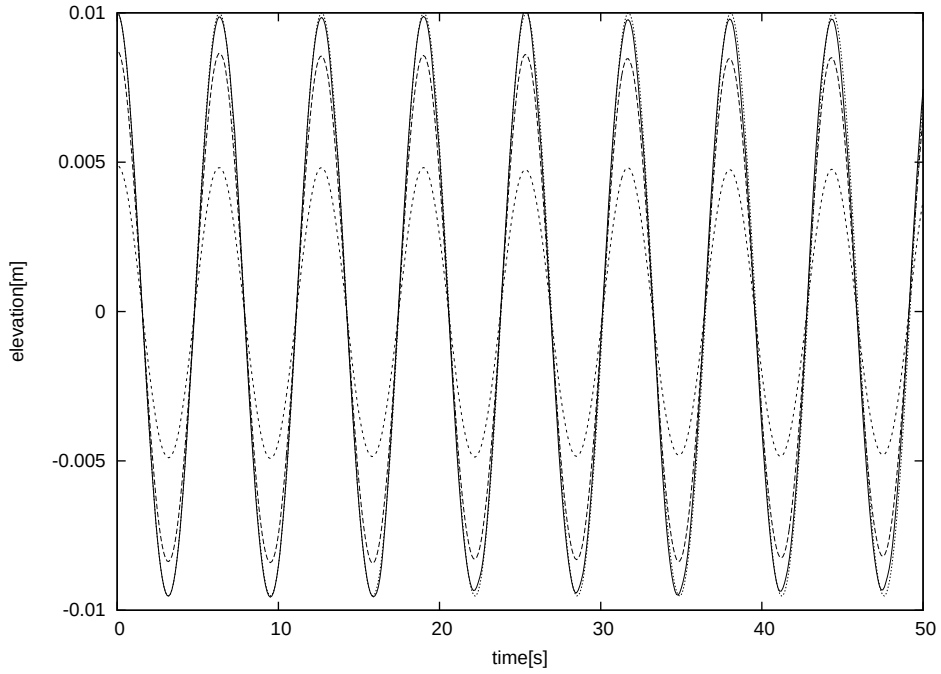

Abbildung 10: *Die Bewegung der freien Oberfläche in drei Positionen, für $y = 0$m und zwei dezentrale Punkte $x = -2$m und $x = -1$m (gestrichelt), wie auch für den zentralen Punkt mit $x = 0$m (durchgezogen). Für die zentrale Position $(x, y) = (0, 0)$m wird die analytische Lösung dargestellt (gepunktet).*

Man erhält eine sehr gute Übereinstimmung von der Periode und der Amplitude der Bewegung verglichen mit der Theorie. Jedoch verringert sich nach 7 Perioden die Amplitude der Bewegung auf ca. 98% des Ausgangswertes, was – neben der numerischen Diffusion bei dem semi-impliziten Faktor $\theta = 0.65$ – vor allem auf die Unvollkommenheit der Reproduzierung eines axialsymmetrischen Falles mit quadratischen Netz-Polygonen zurückzuführen ist.

4.2 Performance

Das für Performance-Tests verwendete Computersystem ist ein Standard-PC ausgestattet mit einem 4-Kern-Prozessor Intel Xeon CPU X5570, 2.93GHz mit 8MB Cache, arbeitend im Tandem (über die PCIe-Schnittstelle) mit einer handelstypischen Grafikkarte GeForce GTX 480 mit 1,4 GHz Shader-Frequenz, 1536 MB GDDR5-Speicher (384 Bit Busbreite, 196.5 GB/s Speicherbandbreite), 15 Multiprozessoren und 480 *Stream Processing Units*, auch *Shader* genannt. Die Grafikkarte hat die CUDA Computing-Fähigkeit 2.0. Ausgestattet mit dem Nvidia Fermi-Chip (GF100), ermöglicht sie Berechnungen in doppelter Genauigkeit ohne allzu große Leistungseinbußen (2× langsamer) verglichen mit einfacher Genauigkeit, die typisch für die Vorgänger-GPUs von Nvidia waren (z.B. GT200-Serien).

Abbildung 11: *Die handelsübliche Grafikkarte Nvidia GeForce GTX480. Bildnachweis: Nvidia Corporation*

Mit dem Hauptinteresse, die rechnerische Leistung des Rechenkerns alleine zu beurteilen, werden die erreichten Geschwindigkeitssteigerungen (Speedups) nur für die Zeitschleife des Algorithmus (Kap. 3.1) angegeben.

Die Ausgabe von Berechnungsergebnissen und alle Initialisierungen in der Ausführungszeit sind nicht mit inbegriffen. Es wird durchgehend das $upwind$-Verfahren Gl. (27) für die Advektionsterme verwendet. Die Vergleiche ignorieren die verstrichene Zeit für die Initialisierung der GPU (für diesen Typ ca. 1.5s). Mit ca. 1.5GB Speicher von der GeForce GTX 480 sind Auflösungen von 4,5 Millionen Zellen in der doppelten und 9,5 Millionen Zellen in der einfachen Genauigkeit möglich.

Die Ergebnisse sind in Tabelle 1 präsentiert und betreffen und betreffen unter der Anwendung von vier Testfällen erreichte Speedups. Die Testfälle sind zwar relativ einfach und schematisch, berücksichtigen aber alle typischen Merkmale des untersuchten Schemas:

1. *Nonlin* – der nicht-lineare Testfall beschrieben im obigen Abschnitt 4.1, berechnet aber implizit ($\theta = 1$) und unter Berücksichtigung der Viskosität und Bodenreibung.

2. *Sloshing* – eine stehende Flachwasserwelle kleinerer Amplitude in einem rechteckigen

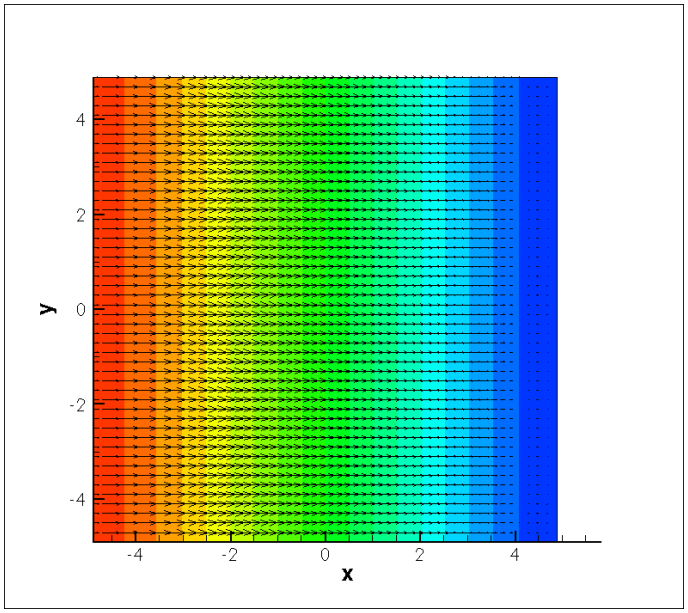

Abbildung 12: *Testfall Sloshing*

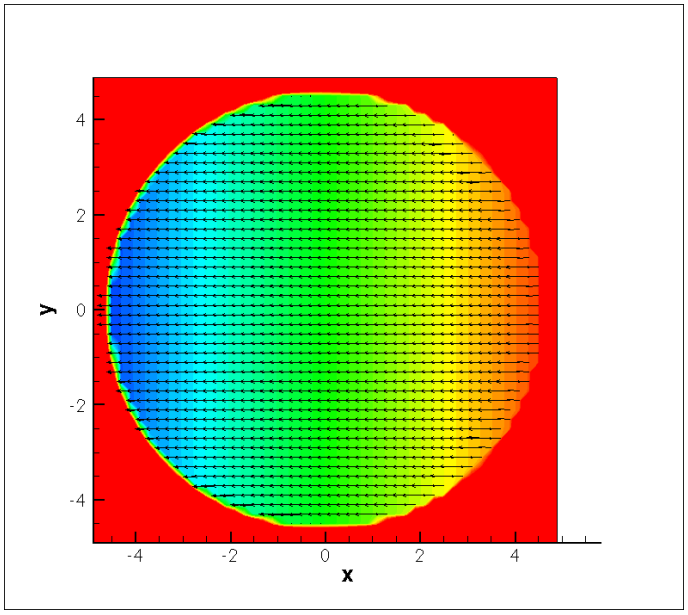

Abbildung 13: *Testfall Lake, einer der ersten Zeitschritten.*

Tabelle 1: *Performance des 2DV-Schemas.*

Beispiel	*Präzision*	*Auflösung*	*Speedup*
(1) *Nonlin*	*einfach*	1000×1000	27
		2000×2000	29
		3000×3000	28
(1) *Nonlin*	*doppelt*	1000×1000	21
		2000×2000	20
(2) *Sloshing*	*einfach*	1000×1000	32
		2000×2000	32
		3000×3000	30
(2) *Sloshing*	*doppelt*	1000×1000	23
		2000×2000	22
(3) *Lake*	*einfach*	1000×1000	32
		2000×2000	34
		3000×3000	33
(3) *Lake*	*doppelt*	1000×1000	24
		2000×2000	23
(4) *Waves*	*einfach*	1000×1000	30
		2000×2000	43
		3000×3000	42
(4) *Waves*	*doppelt*	1000×1000	20
		2000×2000	20

Becken mit einem anfänglich flachen, geneigten Wasserstand (d.h. kein Überflutung und Trockenwerden), Abb. 12.

3. *Lake* – ähnlich wie *Sloshing*, aber in einem Becken mit paraboloidalen Boden, diesmal also mit Berücksichtigung von Überfluten und Trockenfallen. Im Gegensatz zu *Nonlin* die anfängliche freie Oberfläche ist flach, aber geneigt und die Wassertiefen sind auch größer, Abb. 13.

4. *Waves* – Wellen generiert durch eine Wasserstand-Anfangsbedingung – einer Wölbung der freien Oberfläche – greifen die Küsten einer Insel in der Form eines Gauß-Hügels mit Überfluten, Trockenfallen und Reflexionen der Wellen von der Berandung des rechteckigen Berechnungsbereichs, Abb. 14.

In allen Fällen bedeckt das gleiche, quadratische Netz den rechteckigen Berechnungsbereich mit horizontalen Abmessungen 10m $\times$ 10m und mit Gesamtwassertiefen für die Fälle (2-4) von ca. 1m. Der verwendete Zeitschritt Δt variiert zwischen 0.001s und 0.005s; es werden immer 100 Zeitschritte ausgeführt. Die Auflösung $\Delta x = \Delta y$ kann aus der Tabelle 1 abgeleitet werden und variiert zwischen 0.0033m und 0.01m. Die Genauigkeiten des Lösers

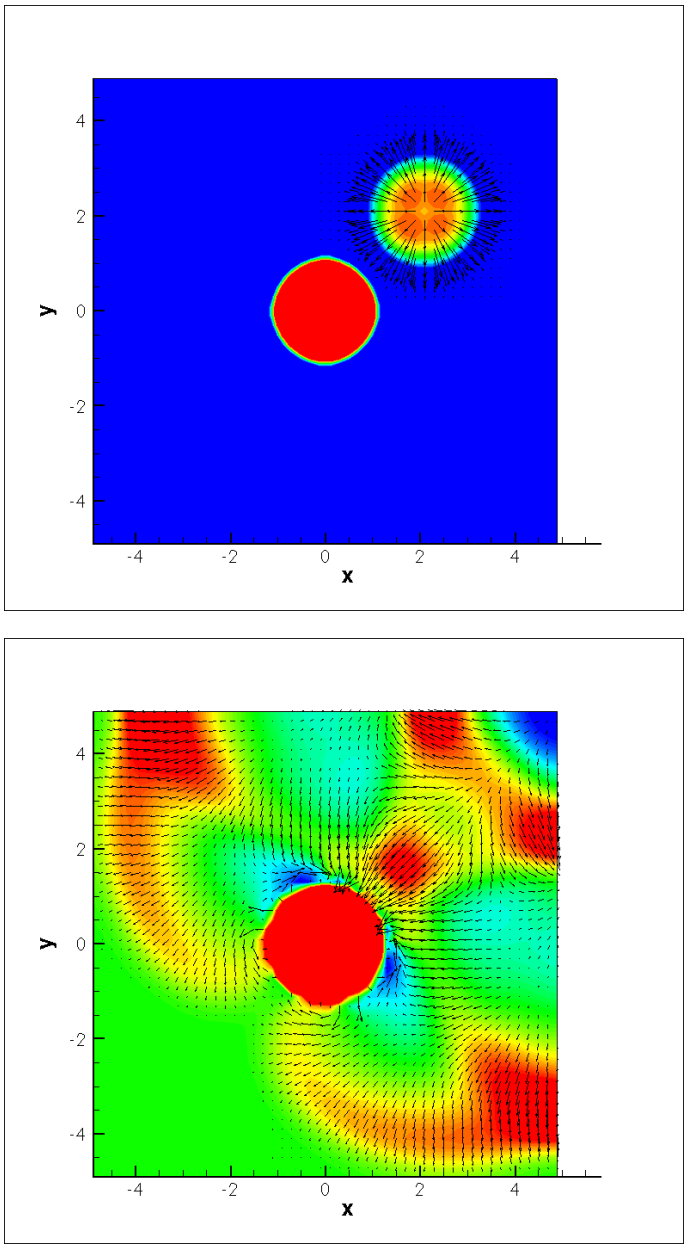

Abbildung 14: *Testfall Waves, Zustand bei einem der ersten Zeitschritte und bei bereits vollständig entwickelten Wellen.*

für die freie Oberfläche (Newton/CG) werden genau wie bei dem oben beschriebenen Verifizierungstestfall gesetzt (Kap. 4.1), der Implizit-Faktor ist $\theta = 1$. In allen Fällen wird ein relativ kleiner Koeffizient für die Bodenrauheit nach Manning 0,001 $[-]$ und eine kleine horizontale Viskosität $\nu = 10^{-4}\mathrm{m}^2/\mathrm{s}$ vorgegeben.

Die erreichten Geschwindigkeitssteigerungen der Ausführung werden auf die Ausführung des gleichen Codes auf *einem Kern* der CPU bezogen. Dies ist der sog. *klassische Speedup*, deren Verwendung in den Leistungsvergleichen zwischen CPUs und GPUs oft scharf kritisiert wird (z.B. [4]), da man dadurch nicht alle Möglichkeiten einer *multicore*-CPU nutzt bzw. einen nur für GPUs optimierten Code verwendet. Für das gegebene PC-System mit einer CPU mit 4 Kernen kann man aber keine Geschwindigkeitssteigerungen mit OPENMP oder MPI über den Faktor 4 erwarten; durch diesen Faktor kann man die erreichten *klassischen* Speedups dividieren, um *faire* Vergleiche GPU contra CPU zu machen. Die rudimentären Vergleiche zwischen einen für CPU-Prozessor-Cache besser optimierten Referenzcode mit doppelten Indizen für die Addressierung der Variablen und den vektorisierten GPU-orientierten Code mit Einzel-Indizen und Verwendung von THRUST und CUSP zeigten kleine Einbußen in den Ausführunszeiten in der Großenordnung von 5%. Es wurden keine OPENMP-Optimierungen an dem Code bzw. an den verwendeten Bibliotheken vorgenommen.

Die erreichten *klassischen* Speedups für die Zeitschleife sind durchweg $O(30)$ für die einfache bzw. $O(20)$ für die doppelte Genauigkeit. Die Speedups $O(40)$ für den *Wellen*-Testfall werden ignoriert, da man ausgerechnet in diesem Fall beobachten konnte, dass der serielle Referenz-Code durch höhere Cache-Misses im vektorisierten Code suboptimal ausgeführt wurde, was nicht näher untersucht wurde. Hiermit liegen die erreichten *fairen* Geschwindigkeitssteigerungen unter rudimentärer Berücksichtigung der Möglichkeiten einer *multicore*-CPU in der Größenordnung $O(5)$ bis $O(8)$, was den theoretisch zu erwartenden Beschleunigungen unter Verwendung von GPUs im Vergleich zu CPUs entspricht [3, 4, 5].

4.3 Realistische Fälle

Es wurde eine Erweiterung des Verfahrens entwickelt, welche die Vorgabe natürlicher Topographien bzw. Bathymetrien aus digitalen Gelände-Modellen erlaubt. Dies ermöglicht Tests in einer natürlichen Situation bezüglich der Bodenbeschaffenheit und Randbedingungen. Hiermit werden ohne weitere Kommentare zwei solche Modelle erwähnt, die gezeitenbeeinflusste Guanabara Bucht (bei Rio de Janeiro, Brasilien), ein Standard-Testfall von TRIM-3D, und eine quasi-stationäre Flussströmung, der Rhein bei Wesel-Xanten, Detschland, adaptiert von einem UNTRIM-Modell, Abb. 15.

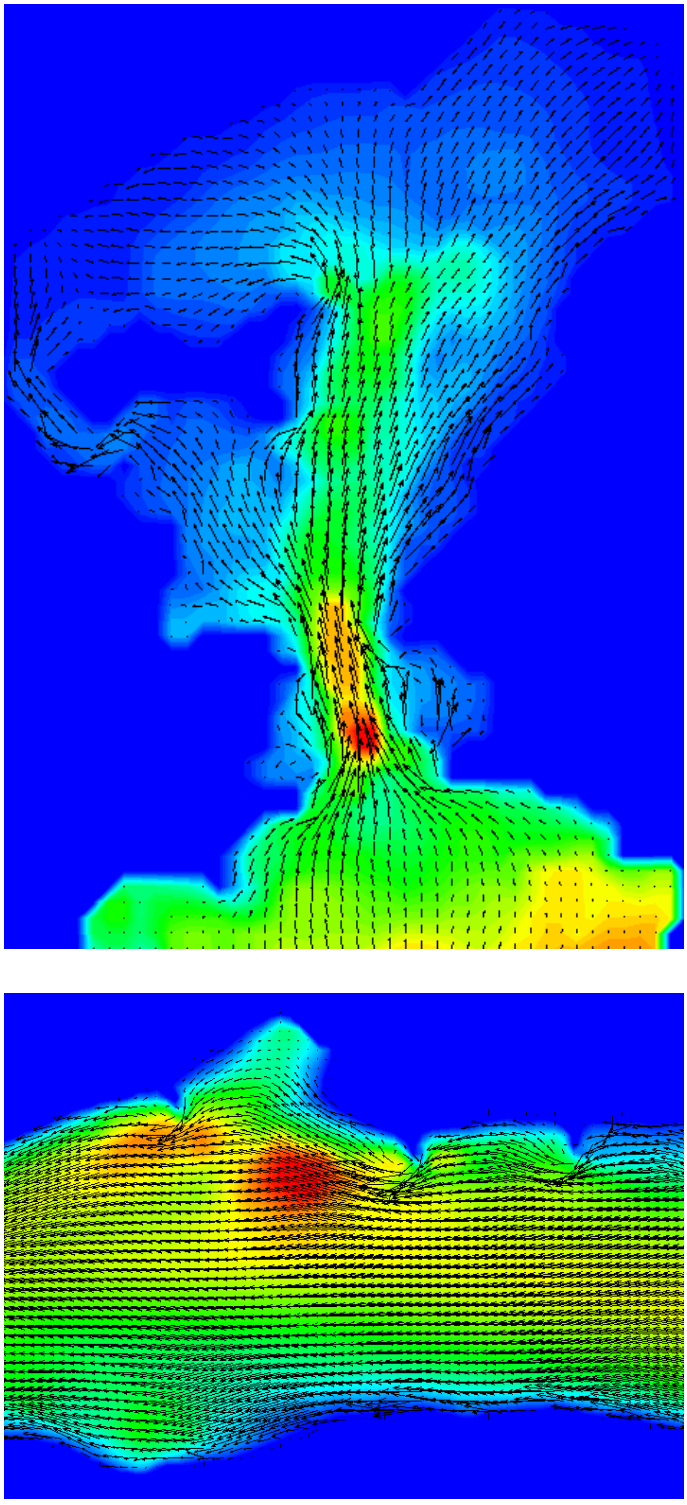

Abbildung 15: *Modelle für natürliche Bathymetrien. Oben: Adaptiertes* TRIM*-Modell Guanabara Bay. Gezeiten in einer Bucht bei Rio de Janeiro. Unten: Adaptiertes* UnTRIM*-Modell Der Rhein bei Wesel-Xanten, quasi-stationäre Strömung in einer Flussstrecke.*

5 Hardware-Adaptierung für existierende Software

5.1 Existierende HPC-Versionen des UnTRIM-Kerns

In diesem Dokument beschäftigen wir uns hauptsächlich mit dem Rechenkern, also mit dem Gleichungslöser für Flachwassergleichungen und nur am Rande mit den spezifischen Aspekten der Vorgabe von Anfangs- und Randbedingungen, Reibungsansätzen, Turbulenzmodellen, Advektionsverfahren, Eingabe/Ausgabe und Datenformaten, etc. Man vergesse aber nicht, dass diese Konzentration auf den Rechenkern nicht bedeutet, dass Anpassungen von der Umgebung des mathematischen Kerns außer Acht gelassen werden können. Weiterhin begrenzen wir uns in diesem Kapitel fast ausschließlich auf die HPC-Aspekte der Implementierung.

UnTRIM besteht aus einem separierten Rechenkern für die numerische Lösung von dreidimensionalen Navier-Stokes-Gleichungen für Strömungen mit freier Oberfläche, einem *User Interface*, das den Zugang zu den Feldern im Rechenkern erlaubt und der sog. *User Library*, die die eigentliche Implementation der Programmsteuerung, Anfangs- und Randbedingungen, Turbulenzmodellierung, Reibungsgesetzen, I/O, etc. beinhaltet [14, 15, 39]. Die Software-Struktur mit einem *User Interface*, das den Rechenkern und die *User Library* trennt, erlaubt völlig unabhängige Entwicklungen – bei den frei programmierbaren Benutzer-Routinen und bei dem getrennten Rechenkern. Es existieren viele Modell-spezifische und Benutzer-spezifische Implementierungen von der *User Library*, aber auch generische, wie im Fall der BAW ProgHome bzw. die Umgebung für die Flussmodellierung in der MPI-Version.

Softwaretechnisch sind die HPC-Anpassungen vom UnTRIM-Rechenkern Abzweigungen von der (Haupt-)Entwicklungslinie von Prof. Casulli, die aber bisher kaum bzw. nie mit seinen Weiterentwicklungen zusammengeführt wurden. Die bisherigen Anpassungen entsprechen in der in Abs. 1.2.1 eingeführten Klassifikation der Parallelität: OpenMP – mittelkörnig und MPI – grobkörnig. OpenMP bringt eine Beschleunigung der Ausführung innerhalb eines Rechen-Knotens bzw. für (seltene) Parallelrechner mit gemeinsamen Speicher (*shared memory*) mit, MPI erlaubt durch die Kommunikation zwischen den separaten Rechen-Knoten Ausführung auf typischen Rechner-Clustern mit getrennten Speichern (*distributed memory*).

OpenMP: Im Auftrag von der BAW wurde UnTRIM-Rechenkern in der Version von 2004 von der Firma Pallas (Dr. Hans-Joachim Plum) mit OpenMP parallelisiert (2003-2004) [54], wobei sehr gute *Speedups* auf Kosten von einigen Veränderungen am Rechenkern erzielt wurden. Diese Version bildete bisher vielleicht das am meisten benutzte Programm für numerische Berechnungen in der Abt. K. Die spätere Versionen von 2007 und insbesondere von 2010 (UnTRIM^2, mit Subgrids, [16, 45]) sind mit weniger Aufwand parallelisiert und liefern dementsprechend kleinere Speedups.

Software-technisch bedeuten OpenMP-Anpassungen im Rechenkern die Modifizierung der eventuell vorhandenen und die Einführung von neuen OpenMP-Direktiven, Veränderungen an dem Code für Reproduzierbarkeit, Genauigkeit und Effizienz bei mehreren

Threads und größeren Datenmengen und vor allem lokale Modifizierungen vom Code für die spezifischen OPENMP-Anforderungen (z.B. Entfernung von parallelen Abhängigkeiten, klare Codierung von komplexeren Schleifen). Die Resultate sind weitgehend von den Eigenschaften der Implementierung von OPENMP in den gegebenen Compiler (vor allem von Intel). Zusätzlich sind sie abhängig von den gegebenen Hardware-Eigenschaften: *shared memory* innerhalb eines Rechen-Knotens bzw. spezifische *shared-memory*-Lösungen für mehrere Knoten, wie SGI NUMA. Es werden aber keine Entwicklungen zusätzlicher Software notwendig, nur lokale, aber manchmal tiefgreifende Anpassungen von diversen Umfang an dem gegebenen Code selbst.

MPI: Die MPI-Anpassungen wurden 2005-2007 realisiert samt aller zusätzlichen Software für die Gebietszerlegung (Kommunikation-Bibliothek, Partitionierung, Zusammenführen der Ergebnisse) in der ursprünglichen Absicht, in der Umgebung von PROGHOME-Anteilen, die als erweiterte UNTRIM *User Library* dienen, verwendet zu werden [39, 40]. Die MPI-Parallelisierung von PROGHOME erfolgte aber nicht und stattdessen wurde in der Abt. W systematisch 2006-2007 eine generische *User Library* gezielt für die hoch auflösende Modellierung von Flussstrecken erstellt. Die Anwendung fand 2007-2012 für ortspezifische Modelle der Abt. W an der Elbe und Donau statt, was gleichzeitig eine umfangreiche und anspruchsvolle Validierung der Software darstellte. 2009 gab es ein Update der MPI-Version. Sie beinhaltet dadurch die nicht-lineare Betrachtung von Überflutung, aber die erst in 2010 eingeführten SubGrids nicht, ein Versions-Update erfolgte bis heute nicht.

MPI-Anpassungen im eigentlichen Rechenkern bestehen aus der Erweiterung der Felder-Spezifikationen um die sog. Halos für die Kommunikation in der Gebietszerlegungsmethode und der gelegentlichen Einführung von Aufrufen für den Datenaustausch zwischen den Prozessen an den richtigen Stellen. In einigen spezifischen und seltenen Fällen (z.B. Initialisierung des Systems) müssen noch Schleifen für erweiterte Feldern angepasst werden. Die gesamte Software dafür ist Eigenentwicklung, deswegen sind die Resultate weitgehend von den verwendeten MPI-Bibliotheken (bisher SGI MPT, Intel, MPICH, OpenMPI), Compilern und Hardware (alle Arten von Clustern) unabhängig. Die MPI-Version des Codes ist natürlich auch seriell (bzw. mit OPENMP-Parallelisierung) ausführbar. Der eigentliche Rechenkern wird also relativ wenig verändert, mit wenigen zerstreuten Modifikationen – die meisten Entwicklungen fanden außerhalb des Kerns statt.

Die **Umgebung des Rechenkerns** – Routinen der *User Interface* (vor allem die `get-` und `set-`Methoden [14]) müssen sowohl für die OPENMP-, wie MPI-Anpassungen leicht angepasst werden, wobei der Umfang für MPI größer (aber unkritisch) ist. Die *User Library* dagegen muss im Fall von MPI auch mit *message-passing* parallelisiert bzw. zur Ausführung mit mehreren Prozessen angepasst werden (insbesondere Eingabe/Ausgabe). Aber: Dies bewirkt, dass sich diese Anteile nicht negativ auf parallele Performance auswirken. Im Fall von OPENMP kann die *User Library* teilweise nicht parallelisiert bleiben, was denn aber entscheidend für die parallele Rechenleistung des Gesamtverfahrens durch die Erhöhung des seriellen Anteils ist.

5.2 Rechenkern für heterogene Rechnerarchitekturen

Die bisherige Praxis, in der man immer aus den neuen Versionen der Entwicklungslinie von Prof. Casulli abzweigt, um einen parallelisierten bzw. Hardware-angepassten Code zu erstellen, wird in Frage gestellt und diskutiert. Als Alternative zu dieser Vorgehensweise steht die Erstellung einer parallelen eigenen Entwicklungslinie der BAW, die HPC-Aspekte stark berücksichtigt, in die danach die Neuentwicklungen aus der Entwicklungslinie von Prof. Casulli eingebaut werden. Die Erstellung einer parallelen Entwicklungslinie erlaubt auch die Möglichkeit eigenhändig Modifikationen an dem Code einzubauen, die aus anderen Quellen als von Prof. Casulli stammen, z.B. bezüglich neuer Advektionsansätze, Netzstrukturen (auch für Subgrids), Ansätze höherer Ordnung, Kopplungen, etc. Hiermit werden die prinzipiellen Varianten der Anpassung für einen heterogenen Parallelrechner diskutiert.

Die Erstellung einer **auf CUDA- bzw. OpenCL-basierenden Version**, die GPUs auf einem Rechenknoten (*standalone* oder im Cluster) nutzt, ist nur mit der Neu-Implementierung des Rechenkerns von UnTRIM bzw. des breiter verstandenen Algorithmus machbar, begleitet durch weitgehende Veränderungen am *User Interface* und der *User Library*. Die Aufwände für einen Code spezialisiert für typische Flussmodellierung sind aber nicht zu groß, um sie nicht in der Eigenentwicklung zu bewältigen. Der wesentliche Vorteil ist die Erzielung einer möglichst optimal beschleunigten Ausführung auf einer GPU bzw. auf einem multi-GPU-Rechen-Knoten. Dies bedeutet aber eine definitive Erstellung einer parallelen Entwicklungslinie zu Prof. Casulli. Hier sollte die Möglichkeit der Verwendung von PGI CUDA FORTRAN erwähnt werden, falls das Verbleiben bei FORTRAN entscheidende Vorteile mit sich bringen kann (z.B. Wiederverwendung existierender Software).

Die Erstellung einer **OpenACC/OpenMP-Version** des Codes, die Mehr-Kern-CPUs und Beschleuniger, wie GPUs bzw. *manycore*-Co-Prozessoren (z.B. MIC) auf einem Rechen-Knoten nutzt, ist wahrscheinlich auf eine weiterhin portierbare Weise bereits 2013 möglich (es wird erwartet, dass OPENACC und OPENMP bald in der Form von OPENMP 4.0 fusionieren werden [51]), ist aber sehr von den kommerziellen Entwicklungen von Dritten und Hardware-Produzenten (hier insbesondere Intel, PGI) abhängig. Der wesentliche Vorteil ist das Versprechen einer breiten Portierbarkeit auf diverse Hardware-Plattformen, aber auf Kosten der parallelen Performance. Dies wäre auch eine Entwicklung, die vermutlich die meisten Chancen hätte, mit der eigenen Entwicklungslinie von Prof. Casulli zusammengeführt zu werden.

Die Fortführung einer **MPI-basierender Version** des existierenden FORTRAN-Codes ist die Voraussetzung für die Nutzung mehrerer Rechen-Knoten in Clustern und dadurch das Erreichen höherer Speedups – als auf einem Rechen-Knoten machbar – möglich. Bisher wurden MPI-Entwicklungen nicht in die Entwicklungslinie von Prof. Casulli integriert. Dies sollte weiterhin eine MPI-Version für Mehr-Kern-CPUs sein (alle Prozesse sind CPU MPI-Prozesse) und als Option eine hybride Version, bei der die MPI-Prozesse zusätzlich mit OPENACC bzw. OPENMP beschleunigt werden. Diese hybride Option könnte attraktive

wirtschaftliche Vorteile mit sich bringen, falls die Ausführung bei voll besetzten Knoten auf diese Weise besser als mit reiner MPI-basierenden Version beschleunigt wäre – d.h. abhängig von der Effizienz der Lösung mit OPENACC/OPENMP. Weiterhin ist MPI-Parallelisierung die effektivste Vorgehensweise, um größere Bibliotheken in der *User Library* zu beschleunigen.

Erstellung einer **CUDA- oder OpenCL-Version, die zusätzlich mit *message-passing* (MPI) parallelisiert wird**. Dieses bedeutet, dass zusätzlich zu dem Rechen-Kern auch eine Neu-Implementierung in neuer Programmiersprache von den ergänzend zum Rechenkern gegebenen Bibliotheken für die MPI-Kommunikation erfolgen müsste. Diese Vorgehensweise ist zur Zeit die sicherste Methode die höchsten Geschwindigkeitssteigerungen auf hybriden CPU/GPU-Clustern zu erreichen. Sie würde eine radikale Veränderung der bisherigen Vorgehensweise bei der Entwicklung bedeuten, hätte aber das Potential, Entwicklungen von Dritten unabhängig von bzw. parallel zu Entwicklungen von Prof. Casulli zu implementieren. Anmerkung: Es gibt bereits erste Entwicklungen, die die Nutzung von mehreren CPU/GPU-Rechen-Knoten im Rahmen von CUDA ohne MPI erlauben, wie die ZeroMQ-Bibliothek [73].

In fast allen Fällen sind Veränderungen am *User Interface* und der *User Library* (in allen Ausführungen) notwendig, wobei im Fall der Erstellung der CUDA- bzw. OPENCL-Version sie am größten sind.

6 Schlussfolgerungen und Empfehlungen

6.1 Vorwort

Wie bereits in Abschnitt 1.4.4 erwähnt, haben während des Projektverlaufes mehrere Veränderungen der Richtung und der Anforderungen bezüglich der Entwicklungsarbeiten an den Modellen für mesoskalige Modellierung in der Abteilung Wasserbau stattgefunden. Angesichts dessen wurde Ende 2012 verlangt, in den Schlussfolgerungen nur zwei Resultate aus diesem Vorhaben zu präsentieren, die für andere Projekte und eventuelle zukünftige Vorhaben eine Bedeutung haben könnten. Dies ist das vollständig für GPUs adaptierte nichtlineare, vertikal gemittelte 2DV-Schema nach Casulli, Kap. 6.2, sowie eine fundierte Beurteilung der Möglichkeiten der Hardware-Adaptierung des Rechenkerns von UnTRIM für die heterogene Parallelrechner, Kap. 6.3. Für die alternativen Advektionsverfahren werden in diesem Umfang keine Schlussfolgerungen bzw. Empfehlungen formuliert.

6.2 Schlussfolgerungen für das 2DV-Schema

6.2.1 Fazit

Es wurde demonstriert, dass eine erfolgreiche und vollständige Adaptierung eines zweidimensionalen numerischen semi-impliziten Verfahrens für Strömungen mit freien Oberfläche für die Hardware-beschleunigte Ausführung auf GPUs möglich ist. Dies ist mit einem relativ geringen Aufwand geschehen und bringt gute Resultate, sogar wenn man sich bei der Suche nach Performance auf gebrauchsfertige numerische Bibliotheken und nicht originäre, spezifische Implementierungen verlässt. Der kritische Punkt für den Erfolg ist, dass man sich nicht nur auf die Anpassung der Performance-kritischen Unterprogramme des Codes konzentriert, sondern versucht, große und zusammenhängende Anteile des Verfahrens ausschließlich auf der Grafikkarte auszuführen. In diesem Fall enthält derart zusammenhängender Bereich die gesamte Zeitschleife des Verfahrens mit der Assemblierung von Matrizen, die Lösung des linearen Gleichungssystems für die Wasserstände eingebettet in den nicht-linearen äußeren Iterationen nach Newton für Überflutungen, sowie alle notwendigen Aktualisierungen von Gesamtwassertiefen und der Geschwindigkeit.

In den Situationen, in denen das rechnerische Gesamtproblem auf die Grafikkarte passt und die Latenzen wegen der Transfers auf/von der Grafikkarte effektiv minimiert werden, sind erhebliche Geschwindigkeitssteigerungen in der Größenordnung von 20 oder 30 jeweils für die doppelte bzw. einfache Genauigkeit verglichen mit einem einzigen CPU-Kern zu erwarten. Dies und die Tatsache, dass für dieses relativ einfache numerische Schema Berechnungen mit ein paar Millionen Zellen auf einer einzigen Grafikkarte möglich sind, öffnen neue Möglichkeiten in der hoch auflösenden Modellierung sogar mit massenproduzierter handelsüblicher Hardware. Für größere Probleme bieten sich Lösungen mit mehreren GPUs bzw. mehreren CPU/GPU-Rechen-Knoten an, die in diesem Projekt wegen der fehlenden Hardware nicht bearbeitet werden konnten.

Allerdings verändert eine solche radikale Adaptierung des Algorithmus die Datenstrukturen für die Datenvektor-orientierten Operationen, was Nachteile für die serielle Ausführung des *gleichen*, d.h. des adaptierten Codes mit sich bringt. Dies erschwert nicht nur die fairen Vergleiche von Geschwindigkeitssteigerungen, sondern auch Wartung eines Codes, der sowohl auf seriellen CPUs, wie auch auf GPUs effizient ausgeführt werden soll.

Einerseits zeigen diese Beobachtungen, dass eine gründliche Re-Programmierung von entscheidenden Teile der *legacy* Codes alleine für die Zwecke der performanten Ausführung auf einer GPU notwendig ist. Auf der anderen Seite ist es klar, dass die Datenvektor-orientierte Programmierung – auch unter Verwendung von numerischen Bibliotheken und mit nachteiligen Effekten für die serielle Ausführung – nachhaltig genug im Hinblick auf kommende neue Computer-Architekturen ist. Hiermit wird die Leistung nicht nur in der massiven Parallelität im Sinne der Anzahl der Prozessor-Einheiten, sondern auch in der Hardware-Beschleunigung von Operationen auf Datenvektoren gesucht. Einige Fragen über die spezifischen Software-Implementierungen bleiben immer noch noch offen, als Programmier-Paradigma sich noch weiterentwickeln und verändern. Diese Unsicherheiten sind aber nicht kritisch für einen kleinen, kompakten Rechenkern mit dem hier gearbeitet wurde.

6.2.2 Ausblick

Die vielversprechenden Ergebnisse für das vertikal integrierte 2DV-Schema suggerieren, dass eine Erweiterung dieser Adaptierung auf drei Dimensionen [10, 13] auch eine gute Geschwindigkeitssteigerung mit sich bringen sollte. Für strukturierte Finite-Differenzen-Netze kann eine Reduzierung der rechnerischen Lasten durch Überspringen der trocken gefallenen (bzw. nie in der Simulation überfluteten) Bereiche durch das Puffern der ausschließlich nassen Netz-Polygone in konsekutive Speicherbereiche erreicht werden [4, 5]. Mit einer spezifischen Vor-Sortierung des Netzes, die die Speicherstruktur einer Grafikkarte berücksichtigt [25], ist eine Adaptierung des äquivalenten Schemas für unstrukturierte Netze (d.h. auch UnTRIM) machbar. Es wird aber an dieser Stelle auf die Vorteile der kartesischen Netze für hochauflösende Modellierung hingewiesen, wie bereits in Abschnitt 1.2.8 beschrieben.

Der entwickelte 2DV-Code ist auch ein nützliches und gebrauchsfertiges Werkzeug zum Testen der Hardware-beschleunigten Implementierungen von weiteren Entwicklungen, wie die vielversprechende SubGrid-Technologie [11, 16, 45], alternative Advektionsverfahren [43, 42, 44, 62], neue Gleichungslöser (z.B. Multigrid-Methoden [60]) andere Netzarten (z.B. Quadtrees, [61]), alternative Ansätze mit gleichem Algorithmus [2], Morphodynamik mit FD-Netzen [7, 46]). Das ideale Einsatzgebiet des resultierenden Codes sind Studien über Überschwemmungen und Trockenfallen in hoch aufgelösten größeren rechnerischen Domänen, die eine ausgezeichnete numerische Effizienz unter Beibehaltung der topographischen Details erfordern.

6.3 Schlussfolgerungen für Hardware-Anpassungen von existierenden Codes

6.3.1 Fazit

Wie in Kap. 5 diskutiert, eröffnen sich in Hinblick sowohl auf das allgemeine Casulli-Verfahren für unstrukturierte Netze, wie auf die existierende Implementation in der Form von UnTRIM mehrere Möglichkeiten für die Hardware-Anpassungen für die Ausführung unter Verwendung von Accelerators. Eine zuverlässige Garantie für die Geschwindigkeitssteigerung der Ausführung und Bewältigung der Berechnungen mit größeren Modellen liefert die Datenparallelität, in diesem Fall mit der Gebietszerlegungsmethode und *message-passing* mittels MPI. Dies sollte die prinzipielle Grundlage für alle weiteren Anpassungen für einen Parallelrechner mit Accelerators sein. Die Beschleunigung der Ausführung auf einem Rechen-Knoten kann mit GPUs bzw. *manycore*-Co-Prozessoren erreicht werden, wobei es zwei sich nicht ausschließende Möglichkeiten der Anpassung gibt. Eine für den existierenden Code relativ konservative Methode ist die Verwendung von OpenACC- bzw. OpenMP-Direktiven, die die weitere Portierbarkeit des Codes verspricht – mit gewissen Performance-Einbußen dafür. Zweite Methode wäre eine Neu-Implementation des Codes mit einer besonders für GPUs spezialisierten Programmiersprache, wie CUDA oder OpenCL, die eine am meisten kontrollierbare Steigerung der Effizienz verspricht – aber diesmal auf Kosten eines gewissen Bruchs mit derzeitigen Entwicklungslinien. Die weiteren Schlussfolgerungen werden in der Form der Empfehlungen für weitere Vorhaben dargestellt.

6.3.2 Empfehlung für weitere Vorhaben

Ausgehend von der derzeitigen Software-Struktur von UnTRIM, bisherigen HPC-Anpassungen und den Erfahrungen aus der Hardware-Adaptierung des o.g. zweidimensionalen Schemas nach Casulli werden hiermit verschiedene, aber sich gegenseitig nicht ausschließende, Möglichkeiten der Adaptierung von UnTRIM im Sinne des Rechenkerns und dessen Umgebung vorgeschlagen.

Es werden im Allgemeinen zwei Wege vorgeschlagen: Einerseits die Fortführung mit dem existierenden Code der wohlerprobten MPI-Parallelisierung basierend auf Datenparallelität als die prinzipielle Quelle der Performance auf einem heutigen Rechner-Cluster ergänzt durch neue Versuche mit OpenACC bzw. OpenMP-Technologie eine Beschleunigung der einzelnen MPI-Prozesse auf Rechenknoten mit GPUs bzw. *manycore*-Co-Prozessoren zu erreichen. Andererseits wird eine alternative Vorgehensweise vorgeschlagen, indem man von Anfang an die feinkörnige und grobkörnige Parallelität im Verfahren hervorhebt und nutzt, um gezielt einen neuen Code für heterogene Parallelrechner z.B. mit CUDA und MPI schreibt. Dies bedeutet, dass man eine Applikation schreibt, die auf *native* Weise die Prozessoren und Co-Prozessoren (Geräte) nutzt, die miteinander mit diversen Methoden kommunizieren können – im Gegensatz zum früher erwähnten *offload*-Modell, in dem die Applikation auf dem (Haupt-)Prozessor ausgeführt wird und nur die ausgewählten Aufgaben auf die Co-Prozessoren bzw. Geräte sendet. Auf diese Weise zeigen sich auch Vorteile von

MPI-Programmen, die mit beiden Ansätzen flexibel strukturiert werden können, sowohl für GPUs wie auch für *manycore*-Co-Prozessoren.

Hiermit werden diese Empfehlungen für weitere Vorghaben konkreter, aber kurz vorgestellt. In allen Fällen wird es stillschweigend angenommen, dass die entsprechende Hardware (z.B. ein Cluster mit CPU/GPU-Rechen-Knoten) und Entwicklungswerkzeuge zur Verfügung stehen.

1. **MPI-Parallelisierung** von der neuesten SubGrid-Version von UnTRIM, bekannt als UnTRIM^2 [16, 45], als die bisher effektivste Methode Geschwindigkeitssteigerung der Ausführung mit der Verwendung eines Clusters. Dies sollte auch kurzfristig erfolgen, mit der zusätzlichen Einführung des NetCDF-Datenformats [72] in *User Library* für die MPI-Version auf gleiche Weise wie BAW-Hamburg und Deltares (CF-Konventionen [71]), um die Post-Processing-Tools beider nutzen zu können, insbesondere für SubGrid-Darstellungen. Dies ist die sicherste und schnellste Weise, die Nutzung von SubGrids in größeren, hoch aufgelösten Modellen sicherzustellen, Entwicklungen in anderen FuE-Vorhaben zu unterstützen und die MPI-Version auf das neueste Niveau der Entwicklungslinie von Prof. Casulli zu bringen. Diese Software ist auch die Grundlage, zumindest als Referenz, für weitere Entwicklungen.

2. ***Offload*-Modell:** OpenACC bzw. OpenMP-Anpassungen (insbesondere mit OpenMP 4.0) von der gleichen neuesten SubGrid-Version von UnTRIM, um das Potential dieser Lösung für einen Multi-CPU-Multi-GPU Rechen-Knoten neu zu untersuchen. Diese Vorgehensweise hat diesen zusätzlichen Vorteil, dass man auch andere Accelerators als (Nvidia) GPUs auf diese Weise berücksichtigt (vor allem Intel Xeon Phi, vel MIC). Es wird 2013 erwartet, dass eine Verschmelzung der beiden Technologien OpenMP und OpenACC stattfinden wird [51]. OpenACC erlaubt bereits viel umfangreichere Kontrolle über die Programmabläufe als z.B. die frühere Accelerator-Technologie von PGI [55], die im Rahmen von diesem Projekt 2010 untersucht wurde und vielversprechend vor allem für diese Codes ist, die bereits früher mit OpenMP parallelisiert wurden.

   Man wird bei diesem Ansatz von Erfolg sprechen, falls die Ausführung mit Beschleuniger schneller sein wird, als mit der rein-CPU-OpenMP und insb. rein-CPU-MPI Version zu erreichen wäre. Die so entstehende Software wird sich problemlos in die existierende Software-Umgebung der BAW Hamburg ProgHome anpassen lassen. In der nächsten Stufe sollte diese Entwicklung mit der o.g. MPI-Parallelisierung für die wahrlich hybride Ausführung integriert werden.

3. ***Native*-Ansatz:** Eine Multi-node-multi-GPU Version mit CUDA oder OpenCL in Verbindung zu MPI zu entwickeln, zuerst mit dem Umfang der *User Library* nur für einfache mesoskalige Flussmodellierung. Das Ziel wäre möglichst vollständige Anwendung bisher bester Methoden, um die neuen Hardware-Architekturen zu nutzen. Zusätzlich wäre bei solcher radikaler Vorgehensweise die Nutzung der besten und aktuell verfügbaren und einfach implementierbaren numerischen Schemata für den allgemei-

nen Casulli-Algorithmus möglich: Neue Netzstrukturen, Advektion- und Diffusion-Schemata [2], Kopplungen mit dem Transport [8], etc. Diese Vorgehensweise wäre möglicherweise bei Beibehaltung der Open-Source-Philosophie auch für andere als bisherige Partner aus Forschung und Industrie attraktiv.

4. ***User Interface & Library*** Weitgehende Anpassung der notwendigsten Anteile der bisher unabhängig entwickelten *User Library* (am besten begleitet mit der Umgebung für Pre- und Post-Prozessing, die auf modernen Datenformaten basiert), die sich mit ersten zwei o.g. Lösungen und vielleicht auch der dritten Vorgehensweise problemlos auf einem hybriden Parallelrechner nutzen lässt. Soll alle notwendigen physikalischen Prozesse beinhalten und Kopplungen mit anderen Modulen (Transport, Wellen, Wasserqualität) erlauben. Die bisherige Probleme bei der Parallelisierung so einer vielfältigen Software können bei der konsequenter Anwendung der groben Datenparallelität (d.h. MPI-Parallelisierung) bewältigt werden.

Literatur

[1] N. Bell and Garland M., *Cusp: Generic parallel algorithms for sparse matrix and graph computations*, 2012, Version 0.3.0, http://cusp-library.googlecode.com.

[2] W. Boscheri, M. Dumbser, and M. Righetti, *A semi-implicit scheme for 3D free surface flows with high-order velocity reconstruction on unstructured Voronoi meshes*, International Journal for Numerical Methods in Fluids (2012), published online, doi: 10.1002/fld.3753.

[3] A.R. Brodtkorb, C. Dyken, T.R. Hagen, J.M. Hjelmervik, and Storaasli O.O., *State-of-the-art in heterogeneous computing*, Scientific Programming **18** (2010), no. 1, 1–33.

[4] A.R. Brodtkorb and M.L. Sætra, *Explicit shallow water simulations on GPUs: guidelines and best practices*, Proceedings of XIX Conference on Water Resources CMWR 2012, 2012, University of Illinois at Urbana-Champaign.

[5] A.R. Brodtkorb, M.L. Sætra, and M. Altinakar, *Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation*, Computers and Fluids **55** (2001), 1–12.

[6] M. Brückner, *Aufbau eines Multi-Kern-GPU-Systems und Evaluierung von neuen Programmiermethoden für numerische Modelle*, Bachelorarbeit, Duale Hochschule Baden-Würtemberg, Karlsruhe, 2010.

[7] M. Burkow, *Numerische Simulation strömungsbedingten Sedimenttransports und der entstehenden Gerinnebettformen*, Diplomarbeit, Institut für Numerische Simulation, Universität Bonn, 2010.

[8] A. Canestrelli, M. Dumbser, A. Siviglia, and E.F. Toro, *Well-balanced high-order centered schemes on unstructured meshes for shallow water equations with fixed and mobile bed*, Advances in Water Resources **33** (2010), 291—-303.

[9] V. Casulli, *Semi-implicit finite difference methods for the two-dimensional shallow water equations*, Journal of Computational Physics **86** (1990), 56–74.

[10] ———, *A semi-implicit finite difference method for non-hydrostatic, free surface flows*, International Journal for Numerical Methods in Fluids **30** (1999), 425–440.

[11] ———, *A high-resolution wetting and drying algorithm for free-surface hydrodynamics*, International Journal for Numerical Methods in Fluids **60** (2009), 391–408, DOI: 10.1002/fld.1896.

[12] V. Casulli and E. Cattani, *Stability, accuracy and efficiency of a semi-implicit method for three-dimensional shallow water flow*, Computers Math. Applic. **27** (1994), no. 4, 99–112.

[13] V. Casulli and R.T. Cheng, *Semi-implicit finite difference methods for three-dimensional shallow water flow*, International Journal for Numerical Methods in Fluids **15** (1992), 629–648.

[14] V. Casulli and G. Lang, *Mathematical model UnTRIM, user interface description*, Technical report, Bundesanstalt für Wasserbau, Hamburg, 2004, Version June 2004 (1.0), 96 pp.

[15] ———, *Mathematical model UnTRIM, validation document*, Technical report, Bundesanstalt für Wasserbau, Hamburg, 2004, Version June 2004 (1.0), 78 pp.

[16] V. Casulli and G.S. Stelling, *Semi-implicit subgrid modelling of three-dimensional free-surface flows*, International Journal for Numerical Methods in Fluids **67** (2011), 441—-449, DOI: 10.1002/fld.2361.

[17] V. Casulli and R. Walters, *An unstructured grid, three dimensional model based on the shallow water equations*, International Journal for Numerical Methods in Fluids **32** (2000), 331–348.

[18] V. Casulli and P. Zanolli, *Semi-implicit numerical modelling of non-hydrostatic free-surface flows for environmental problems*, Mathematical and Computer Modelling **36** (2002), 1131–1149.

[19] ———, *Comparing analytical and numerical solution of nonlinear two and three-dimensional hydrostatic flows*, International Journal for Numerical Methods in Fluids **53** (2007), 1049–1062.

[20] D.M. Causon and M. Ingram, *A Cartesian cut cell method for shallow water flows with moving boundaries*, Advances in Water Resources **24** (2001), no. 8, 899–911.

[21] R.T. Cheng, V. Casulli, and J.W. Gartner, *Tidal, Residual, Intertidal Mudflat (TRIM) model and its applications to San Francisco Bay, California*, Estuarine, Coastal and Shelf Science **36** (1993), 235–280.

[22] J. Cohen and M.J. Molemaker, *A fast double precision CFD code using CUDA*, Proceedings of 21st International Conference on Parallel Computational Fluid Dynamics, Moffett Field, California, USA, 2009.

[23] S. Cook, *CUDA programming: a developer's guide to parallel computing with GPUs*, Applications of GPU Computing, Morgan Kaufmann Publishers, 2012, 560 pp.

[24] A. Corrigan, F.F. Camelli, R. Löhner, and F. Mut, *Semi-automatic porting of a large-scale Fortran CFD code to GPUs*, International Journal for Numerical Methods in Fluids **69** (2011), 314—-331, DOI: 10.1002/fld.2560.

[25] A. Corrigan, F.F. Camelli, R. Löhner, and J. Wallin, *Running unstructured grid-based CFD solvers on modern graphic hardware*, International Journal for Numerical Methods in Fluids **66** (2010), 221—-229, DOI: 10.1002/fld.2254.

[26] A. Defina, *Two-dimensional shallow flow equations for partally dry areas*, Water Resources Research **36** (2000), no. 11, 3251–3264.

[27] Deltares, *Delft3D-FLOW: Simulation of multi-dimensional hydrodynamic flows and transport phenomena, including sediments. User manual*, Delft, the Netherlands, 2011.

[28] N. Deußfeld, *Ein Lagrange-Verfahren 2.Ordnung zur Large-Eddy Simulation*, Diplomarbeit, Bundesanstalt für Wasserbau (BAW), Hamburg, 1999.

[29] Bundesanstalt für Wasserbau (BAW), *HN-Verfahren TRIM-2D. Validierungsdokument, Version 2.0*, Technical report, Bundesanstalt für Wasserbau, Hamburg, 1998, Version Mai 1998 (2.0), 55 pp.

[30] M. Griebel and Zaspel P., *A multi-GPU accelerated solver for the three-dimensional two-phase incompressible navier-stokes equations*, Computer Science – Research and Development **25** (2010), no. (1–2), 65—-73, DOI: 10.1007/s00450-010-0111-7.

[31] M. Gómez-Gesteira, A.J.C. Crespo, B.D. Rogers, R.A. Dalrymple, and Dominguez J.M., *SPHysics – development of a free-surface fluid solver – Part 2: Efficiency and test cases*, Computers & Geosciences **48** (2012), 300–307, DOI: 10.1016/j.cageo.2012.02.028.

[32] M. Gómez-Gesteira, B.D. Rogers, A.J.C. Crespo, R.A. Dalrymple, M. Narayanaswamy, and Dominguez J.M., *SPHysics – development of a free-surface fluid solver – Part 1: Theory and Formulations*, Computers & Geosciences **48** (2012), 289–299, DOI: 10.1016/j.cageo.2012.02.029.

[33] J.-M. Hervouet, *Hydrodynamics of free surface flows. Modelling with the finite element method*, Wiley, Chichester, 2007.

[34] J. Hoberock and N. Bell, *Thrust: A parallel Template Library*, 2010, Version 1.3.0, http://code.google.com/p/thrust/.

[35] S. Hoffmann, *OpenMP*, Informatik im Fokus, Springer Verlag, 2008, 162 pp.

[36] N.M. Hunter, P.D. Bates, S. Neelz, G. Pender, I. Villanueva, N.G. Wright, Liang D., R.A. Falconer, B. Lin, Waller S., A.J. Crossley, and D.C. Mason, *Benchmarking 2d hydraulic models for urban flooding*, Water Management **161** (2008), no. WMI, 13–30, doi: 10.1680/wama.2008.161.1.13.

[37] A. Hérault, G. Bilotta, and R.A. Dalrymple, *SPH on GPU with CUDA*, Journal of Hydraulic Research **48** (2010), no. 51, 74–79, DOI: 10.1080/00221686.2010.9641247.

[38] R. Issa and D. Violeau, *Guide to the Spartacus-2D V1P2 code: Lagrangian modelling of two-dimensional laminar and turbulent flows using the SPH method*, Tech. rep., LNHE, R&D Electricité de France, 2006.

[39] J.A. Jankowski, *Mathematical model UnTRIM - MPI version manual*, Technical report, Bundesanstalt für Wasserbau, Karlsruhe, 2008, Version August 2008 (1.0), 55 pp.

[40] ———, *Parallel implementation of a non-hydrostatic model for free surface flows with semi-lagrangian advection treatment*, International Journal for Numerical Methods in Fluids **59** (2009), no. 10, 1157–1179, DOI: 10.1002/fld.1859.

[41] Khronos OpenCL Working Group, *The OpenCL Specification. Version 1.0*, 2011, http://www.khronos.org/opencl/.

[42] O. Kleptsova, J.D. Pietrzak, and G.S. Stelling, *On a momentum conservative z-layer unstructured C-grid ocean model*, Ocean Modelling **54–55** (2012), 18–36, DOI 10.1016/j.ocemod.2012.06.002.

[43] O. Kleptsova, G.S. Stelling, and J.D. Pietrzak, *An accurate momentum advection scheme for a z-level coordinate models*, Ocean Dynamics **60** (2010), 1447–1461, DOI 10.1007/s10236-010-0350-y.

[44] S.C. Kramer and G.S. Stelling, *A conservative unstructured scheme for rapidly varied flows*, International Journal for Numerical Methods in Fluids **58** (2008), 183—-212, DOI: 10.1002/fld.1722.

[45] G. Lang, *FuE Vorhaben UnTRIM SubGrid-Topographie, A39550370150*, Abschlußbericht, Bundesanstalt für Wasserbau, Hamburg, 2010, 35 pp.

[46] W. Long, J.T. Kirby, and Z. Shao, *A numerical scheme for morphological bed level calculations*, Coastal Engineering **55** (2007), 167—-180, DOI: 10.1016/j.coastaleng.2007.09.009.

[47] Message Passing Interface Forum, University of Tennessee, Knoxville, *MPI: A Message-Passing Interface Standard Version 2.2*, 2009, http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

[48] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide Version 4.0*, 2012, http://developer.nvidia.com/nvidia-gpu-computing-documentation.

[49] OpenACC Corporation, *The OpenACC Application Programming Interface Version 1.0*, 2011, preliminary document, http://www.openacc-standard.org/.

[50] OpenMP Architecture Review Board, *OpenMP Application Program Interface. Version 3.0*, 2008, http://www.openmp.org/.

[51] OpenMP Architecture Review Board, *OpenMP Application Program Interface Version 4.0 - RC 1, Public Review Release Candidate 1*, 2012, http://www.openmp.org/mp-documents/OpenMP4.0RC1_final.pdf.

[52] R. Patzwahl, J.A. Jankowski, and T. Lege, *Very high resolution numerical modelling for inland waterway design*, Proceedings of the International Conference on Fluvial Hydraulics (River Flow 2008), 2008, Izmir, Turkey, 2008.

[53] Bates P.D. and A.P.J. De Roo, *A simple raster-based model for flood inundation simulation*, Journal of Hydrology **236** (2000), no. 1–2, 54–77.

[54] H. Plum, *Introducing OpenMP in UnTRIM*, Tech. report, Pallas GmbH, Brühl, Germany, 2003, 9 pp. (detailed part in Annex).

[55] The Portland Group, *PGI Accelerator Programming Model for Fortran & C, Version 1.3*, 2010, http://www.pgroup.com/resources/accel.htm.

[56] The Portland Group, *CUDA Fortran Programming Guide and Reference, release 2013*, 2013, http://www.pgroup.com/resources/cudafortran.htm.

[57] G. Rosatti, D. Cesari, and L. Bonaventura, *Semi-implicit, semi-Lagrangian modelling for environmental problems on staggered Cartesian grids with cut cells*, Journal of Computational Physics **204** (2005), no. 1, 353–377.

[58] G. Rosatti, R. Chemotti, and L. Bonaventura, *High order interpolation methods for semi-Lagrangian models of mobile-bed hydrodynamics on Cartesian grids with cut cells*, International Journal for Numerical Methods in Fluids **47** (2005), 1269–1275.

[59] J. Sanders, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Longman, Amsterdam, 2010, 312 pp.

[60] R.M. Spitaleri and L. Corinaldesi, *A multigrid semi-implicit finite difference method for the two-dimensional shallow water equations*, International Journal for Numerical Methods in Fluids **25** (1997), 1229–1240.

[61] G.S. Stelling, *Quadtree flood simulations with sub-grid digital elevation models*, Water Management **165** (2012), no. WM10, 567—-580, doi: 10.1680/wama.12.00018.

[62] G.S. Stelling and S.P.A. Duinmeijer, *A staggered conservative scheme for every Froude number in rapidly varied shallow water flows*, International Journal for Numerical Methods in Fluids **43** (2003), 1329–1354.

[63] W.C. Thacker, *Some exact solutions to the non-linear shallow water equations*, Journal of Fluid Mechanics **107** (1981), 499–508.

[64] E.F. Toro, *Riemann solvers and numerical methods for fluid dynamics: A practical introduction*, Springer Verlag, 2009, 3rd. ed., 724 pp.

[65] E.F. Toro and P. Garcia-Navarro, *Godunov-type methods for free-surface shallow flows: A review*, Journal of Hydraulic Research **45** (2007), no. 6, 736–751.

[66] D. Violeau, *Fluid Mechanics and the SPH Method. Theory and Applications*, Oxford University Press, 2012, 616 pp.

[67] J.P. Wang, A.G.L. Borthwick, and R. Eatock Taylor, *Finite-volume type VOF method on dynamically adaptive quadtree grids*, International Journal for Numerical Methods in Fluids **45** (2004), 1–22.

[68] *Intel Compilers web site*, http://software.intel.com/en-us/intel-compilers/.

[69] *CAPS OpenHMPP web site*, http://www.caps-entreprise.com/openhmpp-directives/.

[70] *TOP500 Supercomputer Sites*, http://www.top500.org.

[71] *Deltares NetCDF Website*, http://publicwiki.deltares.nl/display/NETCDF/netCDF.

[72] *Unidata NetCDF web site*, http://www.unidata.ucar.edu/software/netcdf/.

[73] *ZeroMQ, The Intelligent Transport Layer, web site*, http://www.zeromq.org/.