Adaption und Implementierung eines 3D-PIV Algorithmus auf massiv paralleler Hardware

Implementation of a 3D PIV Algorithm on massively parallel Hardware

Kallweit, S.[1,2]; Tedjasukmana, O. [1,2]; Korculanin, O.[2]
1) FH Aachen, Fachgebiet Mess- und Automatisierungstechnik
2) ILA GmbH, Jülich

Rekonstruktionstechniken, 3D-PIV, GPU, Parallele Programmierung
Particle Reconstruction, 3D-PIV, GPU, Parallel Programming

Zusammenfassung

Die Beschleunigung von rechenintensiven 3D-PIV Algorithmen kann mittels kostengünstiger Grafikkarten durchgeführt werden. In diesem Zusammenhang müssen spezielle Eigenschaften der massiv parallelen Hardware von GPUs berücksichtigt werden, um eine entsprechend hohe Steigerung der Rechenleistung zu erzielen. Der verwendete Algorithmus basiert im Wesentlichen auf einer Weiterentwicklung des in [1] eingesetzten Ray-Tracing Verfahrens mit photogrammetrischer Rückprojektion [2]. Verschiedene Verfahrensweisen bei der Implementierung des Algorithmus ermöglichen eine erhebliche Beschleunigung: so wird z.B. der Datenaustausch zwischen Host und GPU minimiert und die Parameter des verwendeten Pinhole Models, die bei jedem detektierten Partikel zur Rekonstruktion benötigt werden, im „Constant Memory“ der GPU für einen schnellen, rein lesenden Speicherzugriff hinterlegt. Weiterhin ist die Nutzung von zusammenhängenden Speicherbereichen (Memory Coalescing) zwingend notwendig. Die Nutzung der CUFFT Bibliothek beschleunigt die FFT-basierte Korrelation beträchtlich. Unter Berücksichtigung einer Vielzahl von Möglichkeiten der GPU Programmierung, konnte der vorliegende Algorithmus nahezu um den Faktor 200 - im Vergleich zur rein seriellen CPU Implementierung - beschleunigt werden.

Einleitung

Die Evaluierung von 3D-PIV Daten ist speicher- und rechenintensiv und benötigt daher entsprechende Ressourcen. Relativ kostengünstige Grafikkarten, die im Vergleich zu herkömmlichen CPUs deutlich höhere Rechenleistungen bei der einfach genauen Fließkommaarithmetik bieten, werden seit einigen Jahren zur Beschleunigung von diversen rechenintensiven Algorithmen, wie z.B. der Computer- oder der Magnetresonanztomographie eingesetzt [3]. Bei der Implementierung der Algorithmen müssen jedoch spezielle Eigenschaften der GPUs berücksichtigt werden, da nur so eine Performancesteigerung im Vergleich zu einem nicht-parallelisierten CPU Code erreicht werden kann.

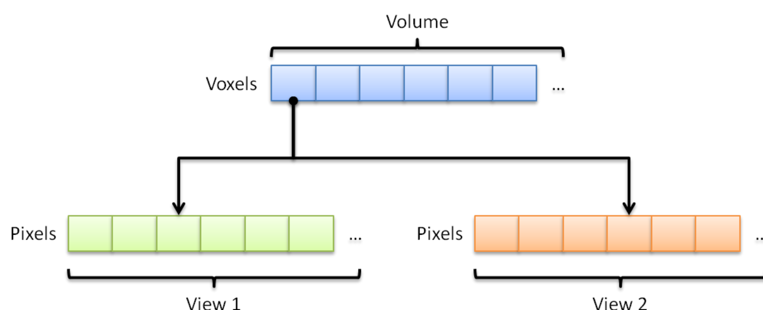

Bild 1: Rückwärtsprojektion des Voxelgrauwertes

Aufbau des Algorithmus

Die beiden wesentlichen Teile des Algorithmus sind die Partikelrekonstruktion und die Bestimmung der Partikelverschiebung mittels 3D-Kreuzkorrelation, Normalisierung und Peak-Detektierung, wie bereits in [1,2] beschrieben. Das Messvolumen wird in Voxel aufgeteilt [1,4], da die 3D-Kreuzkorrelation ein diskretes Gitter benutzt. Die Grauwerte der Voxel werden ähnlich wie in [5] mittels Multiplikation der einzelnen Sichtstrahlen der Kameras berechnet. Ein Schwellenwert, zur Unterdrückung des Hintergrundrauschens, wird festgelegt. Grauwerte unterhalb der Schwelle werden auf Null gesetzt. Diese Unterdrückung (Thresholding) wird beim „strict“ Algorithmus vor der Normalisierung der Grauwerte, die das SNR der späteren Kreuzkorrelation erhöht [6], durchgeführt. Beim „relaxed“ Algorithmus wird das Thresholding nach der Normalisierung implementiert, um Helligkeitsunterschiede der Partikelbilder aus den einzelnen Kameraansichten auszugleichen. Die Rekonstruktion wird vorteilhaft rückwärts, d.h. aus dem betrachteten Volumen heraus, durch Iteration der Voxel durchgeführt.

Implementierung

Die parallele Ausführung von Programmcode wird in CUDA [7] in einem sog. *Kernel* durchgeführt. Die Anzahl der auszuführenden *Threads* wird beim Aufruf des Kernels vorgegeben. Um eine große Anzahl von Threads zu generieren, werden sie in sog. *Blocks* gepackt. *Blocks* können als Felder mit bis zu drei Dimensionen definiert werden. Diese werden schließlich zu einem sog. *Grid*, mit bis zu drei Dimensionen – je nach Compute Capability [7] - zusammengefasst, wie Bild 2 zeigt.

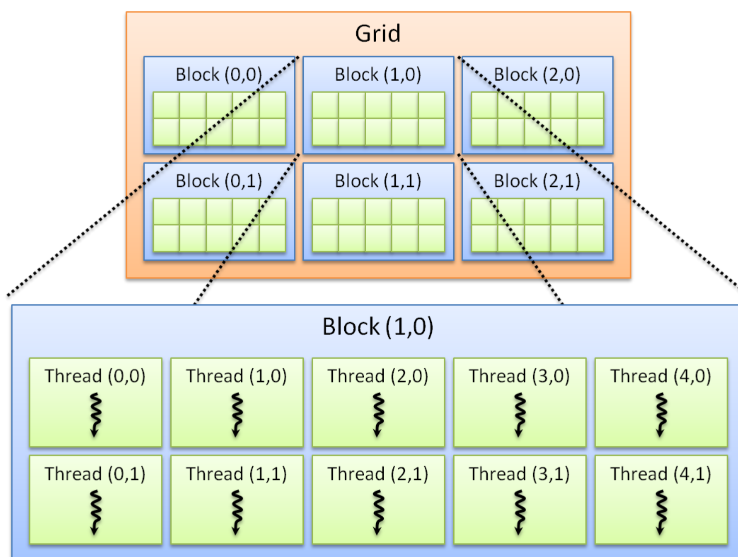

Bild 2: Thread Organisation

Eine triviale Parallelisierung des Algorithmus würde ein Thread pro Voxel vorsehen, welches aber aufgrund der begrenzten Anzahl von Threads und Blocks, die in einem Kernel Aufruf generiert werden können, nicht möglich ist. Die maximale Anzahl der auszuführenden Threads hängt von der Compute Capability ab, die sich bei jeder neuen Generation von GPUs erhöht. Eine Skalierung der auf jedem *Streaming-Multiprozessor* [7] ablaufenden Threads, wird während der Laufzeit des Programmes von CUDA selbständig durchgeführt. Die jetzige Grenze der Threadanzahl wird im Algorithmus durch eine Schleife im Kernel umgangen, deren Iterationsschritt gleich dem Produkt aus Anzahl der Blocks und Anzahl der Threads sein muss (hier: 256 * 32768=8.388.608 Threads).
Eine Speicherung des gesamten zu rekonstruierenden Volumens auf der Grafikkarte ist aufgrund des begrenzten GPU Speichers bei z.B. 1024x1024x512 Voxeln (~2GB bei einem float Datentyp) noch nicht möglich. Da ein Großteil der Voxel nicht mit Grauwerten belegt ist, können Bandspeicher Techniken eingesetzt werden. Jedoch müssen die Daten spätestens vor Ausführung der FFT-Routinen reorganisiert werden. Momentan wird das Volumen in Schnitte

aufgeteilt, die in den GPU Speicher passen, wobei ~60-70% des GPU Speichers belegt werden.

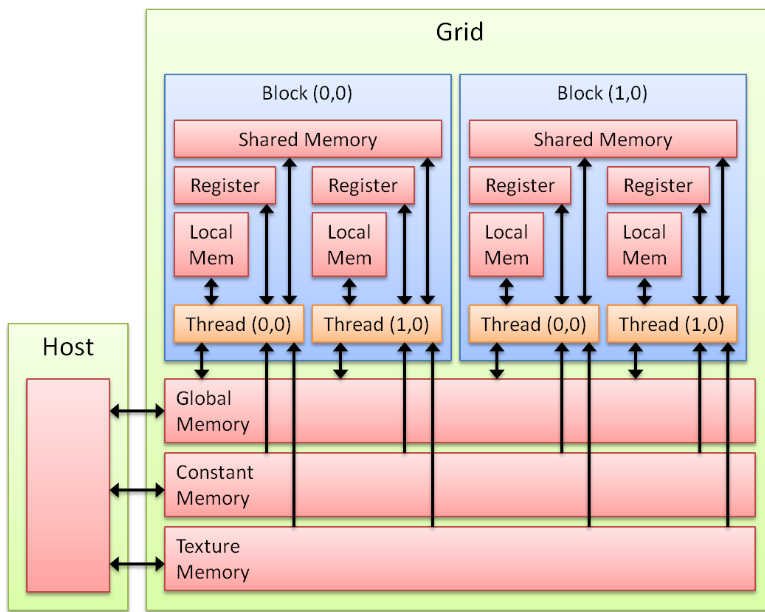

Bild 3: Speicheraufteilung einer Grafikkarte

Die Geschwindigkeit eines parallelisierten GPU Codes hängt wesentlich von der Effizienz und der Art der Speicherzugriffe ab: die Anzahl der Arithmetik Einheiten (ALU) ist so groß, dass – für eine maximale Rechenleistung - jederzeit neue Daten verfügbar sein müssen. Der Speicher einer Grafikkarte ist unterteilt in mehrere unterschiedliche Bereiche wie Bild 3 zeigt. Jeder Thread besitzt einen relativ begrenzten lokalen Speicher in dem kleine Datenmengen abgelegt werden können, auf die der lesende und schreibende Zugriff sehr schnell erfolgt. Jedoch können keine Daten zwischen den Threads unter Benutzung des lokalen Speichers ausgetauscht werden. Dies kann jedoch über den Shared Memory erfolgen, der zumindest in einem Block die Kommunikation ermöglicht. Daten, auf die alle Threads zugreifen müssen, können für Schreib- und Lesezugriffe nur im Global Memory gespeichert werden, wobei er die geringste Bandbreite aufweist. Müssen Daten von allen Threads nur lesbar sein, so können sie im Constant Memory gespeichert werden, der – ausgeführt als Cache Speicher - einen sehr schnellen Zugriff ermöglicht. Hier werden z.B. die Parameter des verwendeten Pinhole Models gespeichert. Die Speicherbereiche Global und Constant Memory sind vom Host durch CUDA Befehle direkt modifizierbar: vor dem Aufruf des Kernels müssen die benötigten Speicherbereiche allokiert und die Initialwerte vom Host in den GPU Speicher übertragen werden. Nach dem Kernelaufruf müssen die Ergebnisse dann zurück in den Host gelesen werden. Da die Hin- und Rückübertragung der Daten Zeit in Anspruch nimmt, sollten nur rechenintensive Algorithmen zur Beschleunigung auf die GPU portiert werden. Weiterhin sollte für sämtliche Rechnungen die GPU benutzt werden, so dass der erneute Rücktransfer der Ergebnisse erst am Schluss erfolgen muss [8].

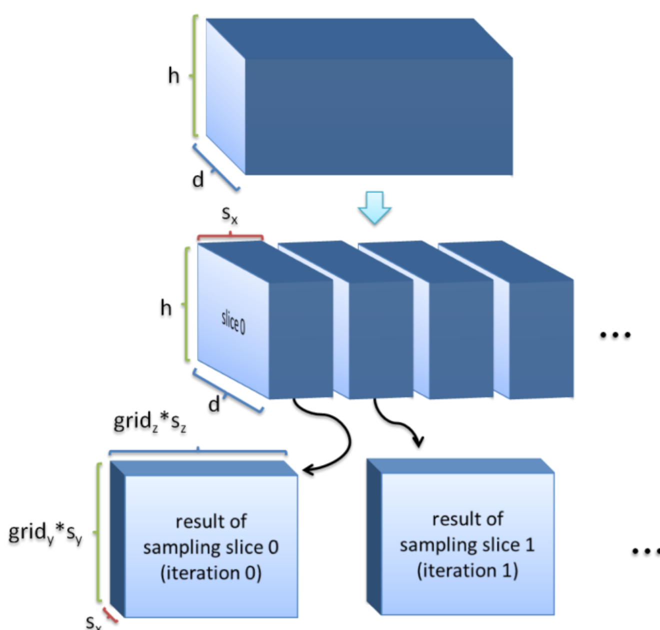

Bild 4: Zerlegung des Volumens in Schichten

Eine günstige Vorgehensweise für den Rekonstruktionsprozess ist:

- Minimierung des Datentransfers zwischen Host und GPU und
- Mapping Parameter, d.h. Pinhole oder algebraisches Modell, in den Constant Memory speichern.

Die 3D-Kreuzkorrelation basiert auf FFT Algorithmen, die die parallele Architektur der GPU zur Erhöhung der Rechenleistung nutzen. Die in CUDA vorhandene Bibliothek CUFFT basiert auf der bekannten FFTW Implementierung und wurde erweitert um ein „Batch“ Feature, welches die parallele Ausführung von mehreren Transformationen erlaubt. Idealerweise müsste die Anzahl der Interrogation Volumen (IV) der Anzahl der auszuführenden Batches entsprechen, was aufgrund des beschränkten GPU Speichers nicht möglich ist. Es erfolgt eine Zerlegung des Volumens in Schichten, die in mehreren Kernel Aufrufen iterativ durchgeführt wird, wobei eine Synchronisation des zu startenden Kernels mit dem gerade abgearbeiteten erfolgen muss, da Kernelaufrufe immer asynchron sind.

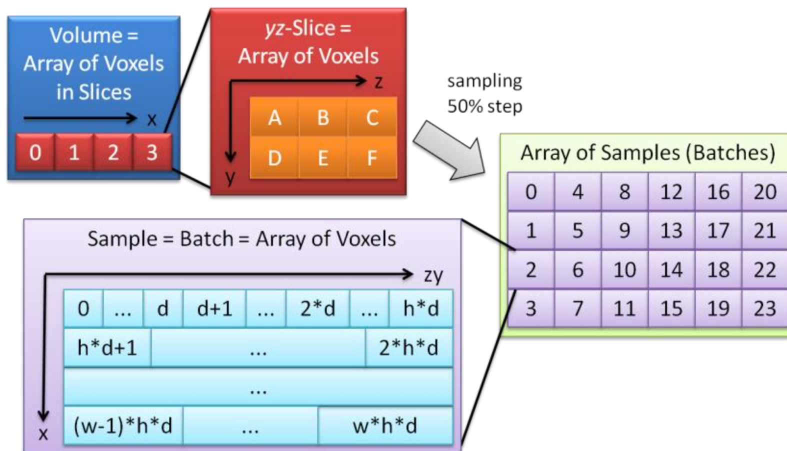

Bild 5: Batch Initialisierung

In einer Iteration wird jede Schicht in die einzelnen Interrogation Volumen aufgeteilt und als Batch hintereinander geordnet (Bild 5). Die spezielle Reihenfolge muss für die Nutzung der CUFFT Routinen eingehalten werden. Die vorhandene CUDA-3D-Kopierfunktion unterstützt sowohl diese Anordnung, als auch den Datentypen nicht, so dass ein weiterer Kernel für die Batch-Initialisierung benutzt wird. Dieser berechnet zusätzlich den Mittelwert und die Standardabweichung der im IV vorhandenen Grauwerte zur späteren Normierung. Hierbei werden sog. *Reduktionsalgorithmen* eingesetzt, die den Shared Memory zur Beschleunigung benutzen. Die benötigte Batch-Identifikation wird erzeugt, in dem jeder Block einem Batch entspricht und jeder Thread nur Elemente in einem Batch bearbeitet.
Nach der Initialisierung werden die Batches mithilfe der CUFFT Bibliothek transformiert. Anschließend werden die konjugiert komplexen Produkte des „Base“- und „Cross“-Batches in einem weiteren Kernelaufruf realisiert. Da die Multiplikation keine Batch-Identifikation benötigt, werden die Daten auf Threads und Blocks gleichmäßig verteilt, d.h. jeder Thread bearbeitet zwei Elemente von den „Base“- und „Cross“-Batches. Falls die Anzahl der Elemente die maximale Anzahl der Threads und Blocks überschreitet, wird eine Schleife im Kernel mit der entsprechenden Schrittweite (wieder: 256 * 32768=8.388.608 Threads) implementiert. Das Produkt wird anschließend per CUFFT rücktransformiert. Die Normierung der Korrelationskoeffizienten erfolgt mit Hilfe der vorher berechneten Standardabweichungen in einem erneuten Kernel, der wiederum die Batch-Identifikation nach dem in der Initialisierung erläuterten Schema durchführt.
Für die Peak Detektion und das Peak Fitting müssen die Koeffizienten reorganisiert und das Maximum ermittelt werden, welches ebenfalls mit einer Reduktion unter Benutzung des Shared Memory erfolgt, wie Bild 6 an einem Beispiel mit acht Threads zeigt.
Das gesamte Post Processing, d.h. die Filterung und Interpolation der Outlier, wird auf dem Host durchgeführt, da der Filteralgorithmus (Global Histogramm [6]) aufgrund zahlreicher if-Anweisungen schlecht parallelisierbar ist.

Optimierte Speichernutzung

Bei den Reduktionsalgorithmen werden redundante Speicherzugriffe, die sonst im Globalen Memory erfolgen würden, mittels des *Shared Memory* beseitigt. Hierbei können Bankkonflikte durch eine sequentielle Adressierung vermieden werden [8].

Bild 6: Beispiel eines Reduktionsalgorithmus für max Operator

Bei der Nutzung des *Global Memory* muss auf die Erzeugung von zusammenhängendem Speicher geachtet werden (Memory Coalescing), um die Bandbreite zu erhöhen. Hierbei muss z.B. die Größe des verwendeten Datentyps berücksichtigt werden: der Datentyp *cufftComplex* (8 Bytes) erfüllt diese Anforderung als Teiler für einen 32, 64 oder 128 Byte Speicherzugriff [9].
Zusätzlich muss die Anzahl der Daten in x und y Richtung der 3D-FFT ein Vielfaches der sog. Warp Size sein, mit einer Größe von z.B. 32 Elementen. Dies wird durch die Begrenzung der IV-Größe auf 32, 64, 96 und 128 Voxel erreicht. Schließlich sollten fortlaufende Threads auf den Speicher auch fortlaufend zugreifen, welches durch eine geeignete Kernelkonfiguration realisiert werden kann. Das Memory Coalescing spielt eine wesentliche Rolle bei der Beschleunigung des Algorithmus. Bei Missachtung kann die Rechenleistung in Abhängigkeit der Datengröße sehr gering werden.

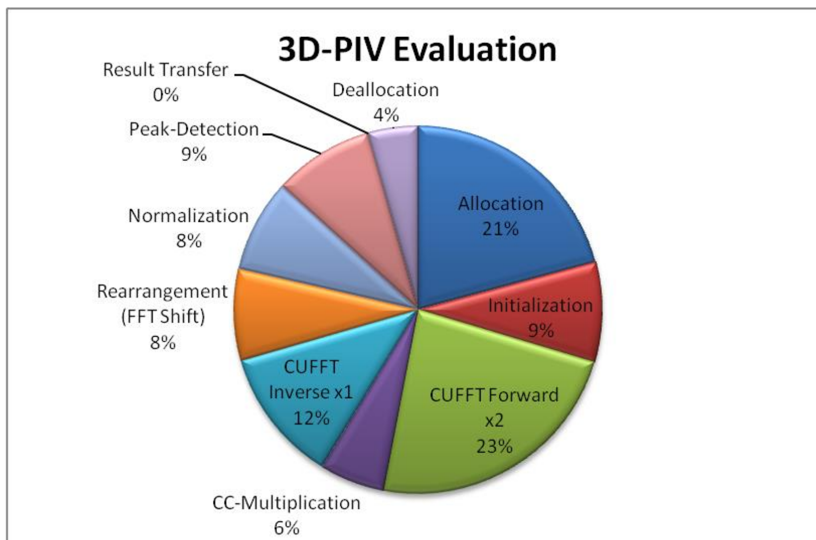

Bild 7: Aufteilung der Rechenzeit bei der 3D PIV Auswertung

Betrachtet man die einzelnen Module für die 3D-PIV Auswertung, so benötigen die 3D-FFT Routinen im Vergleich „nur“ 35% der Gesamtzeit, wohingegen z.B. die Allokierung und Initialisierung der Batches in derselben Größenordnung von ~ 30% liegen. Des weiteren verbrauchen Reduktionsalgorithmen, wie Peak-Detection und Normalization mehr als die Hälfte der rechenintensiven FFT-Routinen. Immerhin 8% der Zeit wird für Reorganisation der Daten verbraucht.

Fazit

Bei „naiver“ Implementierung von Algorithmen auf einer GPU ist die Steigerung der Performance im Vergleich zur rein seriellen Implementierung auf einer Standard CPU meist gering. Nur unter Ausschöpfung einer Vielzahl von recht komplexen Programmierstrategien lassen sich erhebliche Leistungssteigerungen (Bild 8 und Bild 9) erzielen. Ein Vergleich des seriellen Standard CPU Codes mit dem parallelisierten GPU Code zeigte keine nennenswerten Abweichungen der Ergebnisse. Durch die Verwendung des „relaxed“ Algorithmus konnte die Anzahl der Outlier für den Testdatensatz [10] um ca. 11% verringert werden, da die Rekonstruktion bessere Ergebnisse lieferte.

#Voxels	Backward				Forward	
	K	K*	Strict	Relaxed	K*	Strict
73440000	67470 ms	26940 ms	119.36 ms	125.21 ms	10920 ms	106.98 ms
48000000	47810 ms	18110 ms	79.25 ms	82.81 ms	7120 ms	70.78 ms
4608000	1660 ms	1060 ms	8.91 ms	9.61 ms	1020 ms	8.82 ms

Bild 8: Vergleich der Rekonstruktionszeiten für verschiedene Implementierungen (K und K*: serieller Code) bei unterschiedlichen Voxelanzahlen

Setup	K*	Un-normalized	Normalized
E1	119210 ms	660.51 ms	796.46 ms
E2	97860 ms	548.67 ms	661.95 ms

Bild 9: Vergleich der 3D-PIV Auswertezeiten für verschiedene Voxelanzahlen, IV-Größe 64x64x32 (E1: 1200x1000x60, E2: 1000x1000x60)

Literatur

[1] Schimpf, S. Kallweit: Photogrammetric Particle Image Velocimetry, Particle image velocimetry: recent improvements; Proceedings of the Europiv 2 Workshop held in Zaragoza, Spain, March 31 - April 1, 2003 / edited by M. Stanislas, Berlin, Springer 2003. - IX,

[2] S. Kallweit, O. Korculanin, F. Fröhlig, M. Gabi, P. Mattern: Photogrammetric Reconstruction of Particle Positions using Backprojection for 3D-PIV, Fachtagung “Lasermethoden in der Strömungsmesstechnik”, 6. – 8. September 2011, Ilmenau

[3] D. Kirk, W. Hwu: Programming Massively Parallel Processors, Morgan Kaufman, published 2010

[4] Soria, J.: Algebraic Reconstruction Techniques for Tomographic Particle Image Velocimetry, 16th Australasian Fluid Mechanics Conference, Crown Plaza, Gold Coast, Australia, 2-7 December 2007

[5] Atkinson, S. Coudert, J. Foucaut, M. Stanislas, J. Soria: The accuracy of tomographic particle image velocimetry for measurements of a turbulent boundary layer, Experiments in fluids, 2011

[6] Willert, M. Raffel, S. Werely, J. Kompenhans: PIV, A Practical Guide, 2nd Edition, Springer 2007

[7] J. Sanders, E. Kandbrot: CUDA by Example, Addison-Wesley, 2012

[8] Nvidia Corporation: CUDA C BEST PRACTICES GUIDE, DG-05603-001_v5.0, October 2012

[9] Nvidia Corporation: CUDA C PROGRAMMING GUIDE, PG-02829-001_v5.0, October 2012

[10] The Visualization Society of Japan, www. http://www.visualization.jp