Linear Algebra on GPGPUs - II

Suddha Kalyan Basu

Comp 790-058 (Class Lecture)

February 28, 2007

Image: Kolman and Hill, Introductory Linear Algebra, 8^{th} edition

Outline

1. Recap
2. Memory Requirements in Balanced Architectures
3. Sparse Matrix Representations on GPUs
   - Krüger, Westermann
   - Bolz, Farmer, Grinspun, Schröder
4. Conclusions and Summary

Outline

Recap from last lecture..

- Why Linear algebra on GPUs.
  - Parallelizable operations
  - High GPU performance in parallel and streaming computations.
- Matrix Multiplications.
  - CPU and GPU-friendly methods.
- GPU programming
  - CUDA - access to shared memory, block threading.

Outline

Balanced Architectures

Processing Elements (PEs) are characterized by the following:

- Computational bandwidth (C)
- I/O bandwidth (IO)
- Size of local memory (M)

Balanced PE

A PE is *balanced* if the I/O time equals computation time.

Balanced Architectures

Processing Elements (PEs) are characterized by the following:

- Computational bandwidth (C)
- I/O bandwidth (IO)
- Size of local memory (M)

Balanced PE

A PE is *balanced* if the I/O time equals computation time.

Challenges

- Making use technological advances such as high computational bandwidth of CPUs, high I/O bandwidth of GPUs.
- Keeping architectures balanced.

$$\frac{N_C}{C} = \frac{N_{IO}}{IO}$$

N_C, N_{IO} are the total number of operations and word exchanges for a computation, respectively.

- If C/IO increases by α (as when using an array of PEs), N_C/N_{IO} must also increase by the same ratio.
- N_C/N_{IO} is often a function of the size of local memory M.

Challenges

- Making use technological advances such as high computational bandwidth of CPUs, high I/O bandwidth of GPUs.
- Keeping architectures balanced.

$$\frac{N_C}{C} = \frac{N_{IO}}{IO}$$

N_C, N_{IO} are the total number of operations and word exchanges for a computation, respectively.

- If C/IO increases by α (as when using an array of PEs), N_C/N_{IO} must also increase by the same ratio.
- N_C/N_{IO} is often a function of the size of local memory M.

Matrix Multiplication

Multiply two matrices A and B, each of size $N \times N$. Local memory size is M.

- Multiply a $\sqrt{M} \times N$ submatrix of A with $N \times \sqrt{M}$.
- Compute $\sqrt{M} \times \sqrt{M}$ submatrices of the product matrix.

$$
\begin{aligned}
N_C &= \Theta(N \cdot M) \\
N_{IO} &= \Theta(N \cdot \sqrt{M}) \\
\frac{N_C}{N_{IO}} &= \Theta(\sqrt{M})
\end{aligned}
$$

If $\frac{N_C}{N_{IO}}$ increases by α, M has to increase by a factor of α^2.

Matrix Multiplication

Multiply two matrices A and B, each of size $N \times N$. Local memory size is M.

- Multiply a $\sqrt{M} \times N$ submatrix of A with $N \times \sqrt{M}$.
- Compute $\sqrt{M} \times \sqrt{M}$ submatrices of the product matrix.

$$\begin{aligned} N_C &= \Theta(N \cdot M) \\ N_{IO} &= \Theta(N \cdot \sqrt{M}) \\ \frac{N_C}{N_{IO}} &= \Theta(\sqrt{M}) \end{aligned}$$

If $\frac{N_C}{N_{IO}}$ increases by α, M has to increase by a factor of α^2.

Grid Computations

Consider a grid of dimension d, size N^d. Every grid cell is updated with the weighted average of cells in a surrounding window. An array of PEs to perform grid operations, each having memory of size M. Let $l = M^{1/d}$.

- Local memory stores a $l \times \ldots \times l$ subgrid.
- I/O fetches the neighboring elements at boundaries. Size of boundary is l^{d-1}.

$$\begin{aligned} N_C &= \Theta(l^d) = \Theta(M) \\ N_{IO} &= \Theta(l^{d-1}) \\ \frac{N_C}{N_{IO}} &= \Theta(l) = \Theta(M^{1/d}) \end{aligned}$$

If $\frac{N_C}{N_{IO}}$ increases by α, M has to increase by a factor of α^d.

Grid Computations

Consider a grid of dimension d, size N^d. Every grid cell is updated with the weighted average of cells in a surrounding window. An array of PEs to perform grid operations, each having memory of size M. Let $l = M^{1/d}$.

- Local memory stores a $l \times \ldots \times l$ subgrid.
- I/O fetches the neighboring elements at boundaries. Size of boundary is l^{d-1}.

$$
\begin{aligned}
N_C &= \Theta(l^d) = \Theta(M) \\
N_{IO} &= \Theta(l^{d-1}) \\
\frac{N_C}{N_{IO}} &= \Theta(l) = \Theta(M^{1/d})
\end{aligned}
$$

If $\frac{N_C}{N_{IO}}$ increases by α, M has to increase by a factor of α^d.

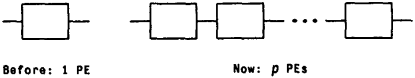
More Results

- FFT:
  $M_{new} = (M_{old})^{\alpha}$
- Matrix Triangularization:
  $M_{new} = \alpha^2 M_{old}$
- Sorting:
  $M_{new} = (M_{old})^{\alpha}$
- Matrix-vector Multiplication, solving triangular linear systems:
  Not possible - system cannot be rebalanced merely by increasing the memory size of PEs.

Implications for Parallel Architectures

- Comparing memory requirements of an architecture with **single-PE** and one with an **array of p PEs.**
- Computational power of the new system is p times that of the old one.
- To maintain a balanced architecture, the parallel system must have a *larger* local memory than the single PE in the original system.

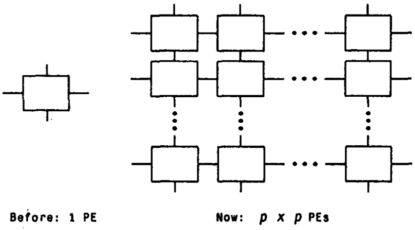
1-D Array of Processors



$$C_{new} = p \cdot C$$
$$IO_{new} = IO$$
$$C_{new}/IO_{new} = p \cdot (C/IO)$$

For scientific computations like matrix multiplication, grid computation and triangularization, $M_{new} = p^2 M$. Thus, **each of the PEs must have a local memory** p **times larger** than the original PE.

2-D Array of Processors



$$
\begin{aligned}
C_{new} &= p^2 \cdot C \\
IO_{new} &= p \cdot IO \\
C_{new}/IO_{new} &= p \cdot (C/IO)
\end{aligned}
$$

To meet the condition $M_{new} = p^2 M$ for the system, **the local memory for each PEs can be independent of** p. Such a system is *automatically balanced*.

For d-dimensional array of processors, computations with the property that $M_{new} = \alpha^d M$ is automatically balanced.

CPU-GPU comparison

CPU- high computational b/w, GPU- high I/O b/w.

If, for some $\beta > 1$,

$$\frac{C_{CPU}}{IO_{CPU}} = \frac{C_{GPU}}{IO_{GPU}} \cdot \beta$$

To perform a given computation with same performance, CPU cache size must be altleast β^2 times larger than the GPU cache size.

Pentium 4 - Cache: 2 MB (single core), 4 MB (Dual Core)
GPU - Cache: 128 KB.
However for 3GHz P4 vs. 7800 GTX, $\beta \approx \frac{3/0.5}{10/100} = 60$.

CPU-GPU comparison

CPU- high computational b/w, GPU- high I/O b/w.

If, for some $\beta > 1$,

$$\frac{C_{CPU}}{IO_{CPU}} = \frac{C_{GPU}}{IO_{GPU}} \cdot \beta$$

To perform a given computation with same performance, CPU cache size must be altleast β^2 times larger than the GPU cache size.

Pentium 4 - Cache: 2 MB (single core), 4 MB (Dual Core)
GPU - Cache: 128 KB.
However for 3GHz P4 vs. 7800 GTX, $\beta \approx \frac{3/0.5}{10/100} = 60$.

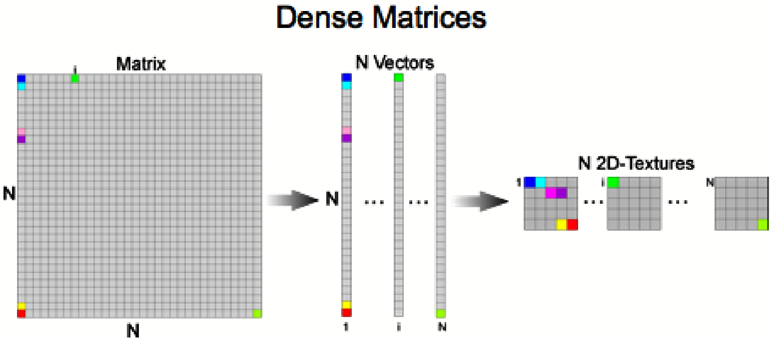
Outline

1. Recap
2. Memory Requirements in Balanced Architectures
3. Sparse Matrix Representations on GPUs
   - Krüger, Westermann
   - Bolz, Farmer, Grinspun, Schröder
4. Conclusions and Summary

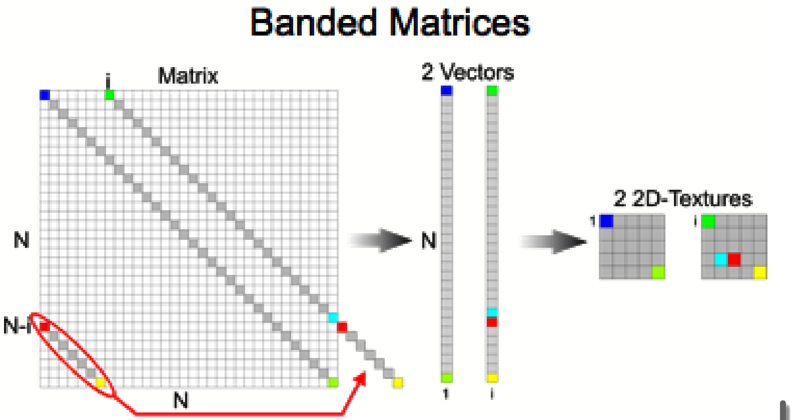
Sparse Matrices: Problems

- Suffers due to random accesses to memory (cache unfriendly).
- Important to represent sparse matrices in a way so that cache misses are reduced.
- Large linear systems often have sparse matrices.
  - Fluid equations, wave equations.

Dense Matrix Representation

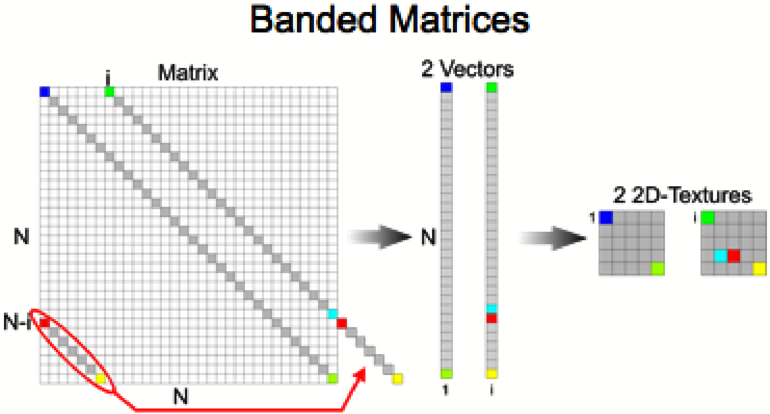


Banded Matrix Representation



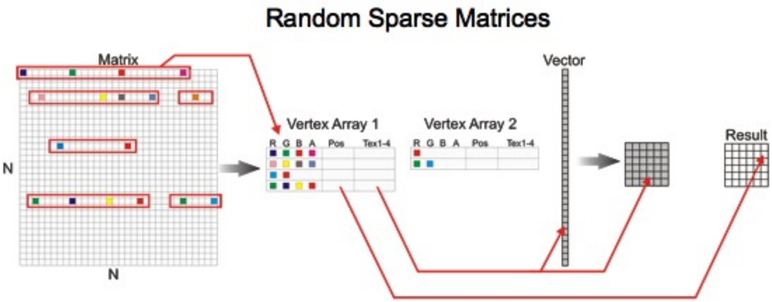
Why do this? Cache Efficiency.

Banded Matrix Representation

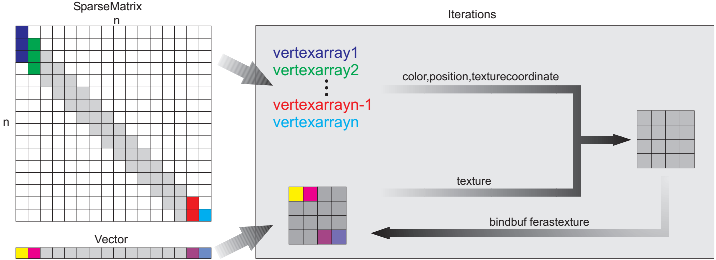


Why do this? Cache Efficiency.

Random Sparse Matrix Representation



Sparse Matrix - Vector Multiply



Conjugate Gradient Method

Unpreconditioned CG

$p^{(0)} = r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
for $i \leftarrow 0$ **to** $\#itr$
  $\rho_i = {r^{(i)}}^T r^{(i)}$
  $q^{(i)} = Ap^{(i)}$
  $\alpha_i = \rho_i / {p^{(i)}}^T q^{(i)}$
  $x^{(i+1)} = x^{(i)} + \alpha_i p^{(i)}$
  $r^{(i+1)} = r^{(i)} - \alpha_i q^{(i)}$
  $\beta_i = {r^{(i+1)}}^T r^{(i+1)} / \rho_i$
  $p^{(i+1)} = r^{(i+1)} + \beta_i p^{(i)}$
  convergence check

Unpreconditioned GPU-based CG

clMatVec$(CL\_SUB, A, x^{(0)}, b, r^{(0)})$ initial guess $x^{(0)}$
clVecOp$(CL\_ADD, -1, 0, r^{(0)}, NULL, r^{(0)})$
clVecOp$(CL\_ADD, 1, 0, r^{(0)}, NULL, p^{(0)})$
for $i \leftarrow 0$ **to** $\#itr$
  $\rho_i =$ **clVecReduce**$(CL\_ADD, r^{(i)}, r^{(i)})$
  clMatVec$(CL\_ADD, A, p^{(i)}, NULL, q^{(i)})$
  $\alpha_i =$ **clVecReduce**$(CL\_ADD, p^{(i)}, q^{(i)})$
  $\alpha_i = \rho_i \,/\, \alpha_i$
  clVecOp$(CL\_ADD, 1, \alpha_i, x^{(i)}, p^{(i)}, x^{(i+1)})$
  clVecOp$(CL\_SUB, 1, \alpha_i, r^{(i)}, q^{(i)}, r^{(i+1)})$
  $\beta_i =$ **clVecReduce**$(CL\_ADD, r^{(i+1)}, r^{(i+1)})$
  $\beta_i = \beta_i \,/\, \rho_i$
  clVecOp$(CL\_ADD, 1, \beta_i, r^{(i+1)}, p^{(i)}, p^{(i+1)})$
  convergence check

Performance

Graphics card used: **ATI 9800**

- Vector-vector multiply:
  - 512^2: 0.2 ms, 1024^2: 0.72 ms, 2048^2: 2.8 ms.
- Dense Matrix-vector:
  - 4096×4096: 230 ms.
- Sparse Matrix-vector:
  - (Banded, 10 non-zero diagonals) 4096×4096: 0.72 ms, (Random) $1024^2 \times 1024^2$: 4.54 ms.

Discussion

- Data resides on GPU memory during all iterations.
  - Possible because matrix A is static.
- Only the final result needs to be passed to the application.
- Considerable speed-up due to use of RGBA texels for storing 4 vector entries.
- Contribution:
  - Vector/Matrix representation
  - Basis linear algebra operators

Discussion

- Data resides on GPU memory during all iterations.
  - Possible because matrix A is static.
- Only the final result needs to be passed to the application.
- Considerable speed-up due to use of RGBA texels for storing 4 vector entries.
- Contribution:
  - Vector/Matrix representation
  - Basis linear algebra operators

Discussion

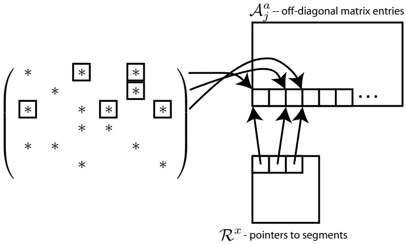
- Data resides on GPU memory during all iterations.
  - Possible because matrix A is static.
- Only the final result needs to be passed to the application.
- Considerable speed-up due to use of RGBA texels for storing 4 vector entries.
- Contribution:
  - Vector/Matrix representation
  - Basis linear algebra operators

Discussion

- Data resides on GPU memory during all iterations.
  - Possible because matrix A is static.
- Only the final result needs to be passed to the application.
- Considerable speed-up due to use of RGBA texels for storing 4 vector entries.
- Contribution:
  - Vector/Matrix representation
  - Basis linear algebra operators

Alternate Sparse Matrix Representation

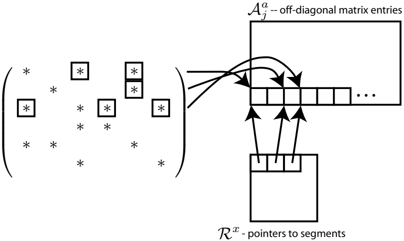
1. Vector $\mathbf{x}$ in texture $\mathcal{X}^x$
2. Matrix A stored in 2 textures: diagonal and off-diagonal non-zero entries separately.
3. Indirection texture $\mathcal{R}^x$.
4. Column indices $\mathcal{C}^a$, laid out exactly as $\mathcal{A}^a_j$, having pointers to corresponding entries in $\mathcal{X}^x$.



Storing diagonal and off-diagonal entries separately help in preconditioning for C-G method.

Alternate Sparse Matrix Representation

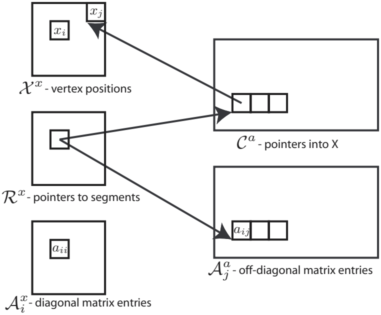
1. Vector $\mathbf{x}$ in texture $\mathcal{X}^x$
2. Matrix A stored in 2 textures: diagonal and off-diagonal non-zero entries separately.
3. Indirection texture $\mathcal{R}^x$.
4. Column indices $\mathcal{C}^a$, laid out exactly as $\mathcal{A}_j^a$, having pointers to corresponding entries in $\mathcal{X}^x$.



Storing diagonal and off-diagonal entries separately help in preconditioning for C-G method.

Computing Matrix Entries

Result of matrix-vector multiplication: $\mathcal{Y}^x$ (texture).

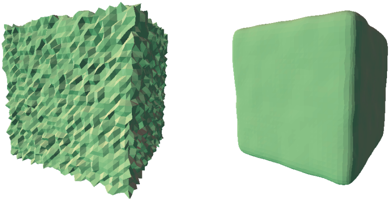


$$
\begin{aligned}
j &= \mathcal{R}^x[i] \\
\mathcal{Y}^x[i] &= \mathcal{A}^x_i[i] * \mathcal{X}^x[i] + \sum_{c=0}^{k_i-1} \mathcal{A}^a_j[j+c] * \mathcal{X}^x[\mathcal{C}^a[j+c]],
\end{aligned}
$$

Optimizations

- Round-robin pipelining of texture access
  - Multithreading: q independent stream records, processed in an interleaved manner.
  - Instructions $I_1, I_2, \ldots$, Records $R_1, R_2, \ldots, R_q$ executed as $I_1(R_1), I_1(R_2), \ldots, I_1(R_q), I_2(R_1), I_2(R_2), \ldots, I_2(R_q), \ldots$.
  - Hides latency between two data-dependent instructions.
- Making use of SIMD execution style:- Choose rectangle area appropriately.
  - p parallel pipelines, q records, choose $w \cdot h \approx p \cdot q$.

Performance



CPU: **3GHz Pentium 4**, GPU: **nVIDIA GeForce FX**

- Unstructured matrix multiplications: (Size: $37k \times 37k$, Avg. non-zero entries per row: 7)
  - CPU: 13.33 ms, GPU: 8.33 ms (theoretical bound: 2 ms)
- Structured matrix multiplications: (Grid size - 257×257)
  - CPU: 1.33 ms, GPU: 0.73 ms (theoretical bound: 0.21 ms)

Outline

Remarks

- Combination of approaches:
  - Multi-threading to hide latency, along with Krüger's representation of sparse matrices.
- Compare performances of CUBLAS on G80, with the above results.

References

1. Krüger and Westermann, ”Linear Algebra Operators for GPU Implementation of Numerical Algorithms.” ACM SIGGRAPH 2003.
2. Bolz, Farmer, Grinspun and Schröder, ”Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid”. ACM SIGGRAPH 2003.
3. Kung, ”Memory Requirements for Balanced Architechtures”, ISCA 1986: Proceedings of the 13th annual international symposium on Computer architectures.