Efficient Gram-Schmidt orthogonalization with CUDA for iterative eigensolvers

Andrés Tomás Vicente Hernández

Grid and High Performance Computing Group
Universidad Politécnica de Valencia, Spain

June 13, 2010

UNIVERSIDAD
POLITECNICA
DE VALENCIA

Grid y Computación
de Altas Prestaciones
GRyCAP

Outline

Standard Eigenproblem

$$Ax = \lambda x, \ A \in \mathbb{C}^{n \times n}$$

n solutions

$$\lambda_1, \lambda_2, \ldots, \lambda_n \in \mathbb{C}$$

$$x_1, x_2, \ldots, x_n \in \mathbb{C}^n$$

Example application areas:

- Dynamic structural analysis (e.g. civil engineering)
- Stability analysis (e.g. control engineering)
- Eigenfunction determination (e.g. electromagnetics)
- Bifurcation analysis (e.g. fluid dynamics)
- Information retrieval (e.g. latent semantic indexing)

Iterative eigensolvers

Iterative methods are

- useful for computing only part of spectrum (i.e. largest or smallest eigenvalues)
- efficient for sparse problems (because matrix elements are not changed)

SLEPc Scalable Library for Eigenvalue Problem Computations

A *general* library for solving large-scale sparse eigenproblems

- Based on PETSc
- Parallel implementation with MPI
- Krylov and Jacobi-Davidson methods

Available at `http://www.grycap.upv.es/slepc`

Orthogonalization in iterative eigensolvers

Used for

- building a Krylov subspace
- deflation against already converged eigenvectors

Iterative Classical Gram-Schmidt is the procedure of choice

- Well-known reorthogonalization criteria (DGKS)
- As stable as Householder but fewer operations
- Implemented with BLAS2 operations

Most expensive part of the solver

- performance is limited by memory bandwidth rather than computing speed
- good candidate for GPU acceleration (because graphics memory have greater bandwidth than main memory)

Outline

Arnoldi iteration

Arnoldi decomposition of size m

$$AV_m = V_m H_m + h_{m+1,m} v_{m+1} e_m^*$$

- Store V and H in graphics memory
- Use CUBLAS for V and H operations
  CUBLAS is the BLAS implementation by nVidia for its own graphics processors
- A matrix vector products are performed in the CPU $\rightarrow$
  two vector transfers per iteration

Arnoldi iteration

for $j = 1, 2, \ldots, m$

copy v_j to the CPU

$v_{j+1} = Av_j$

copy v_{j+1} to the GPU

$h_{1:j,j} = 0$

$h_{j+1,j} = \|v_{j+1}\|_2$ `cublasSnrm2`

$k = 0$

repeat

$\rho = h_{j+1,j}$

$c = V_j^* v_{j+1}$ `cublasSgemv`

$v_{j+1} = v_{j+1} - V_j c$ `cublasSgemv`

$h_{1:j,j} = h_{1:j,j} + c$ `cublasAxpy`

$h_{j+1,j} = \|v_{j+1}\|_2$ `cublasSnrm2`

$k = k + 1$

until $h_{j+1,j} > \eta\,\rho \vee k > 2$

$v_{j+1} = v_{j+1}/h_{j+1,j}$ `cublasScal`

end

Krylov-Schur method

Proposed by [Stewart, 2001] as an equivalent method to IRAM (ARPACK), but easier to compute and more stable

Algorithm

1. Build an initial Arnoldi decomposition of size m
2. Convert to a Krylov-Schur decomposition
3. Extract approximate Ritz values and vectors
4. Exit if convergence is achieved
5. Update and truncate the decomposition to a size $k = m - p$
6. Extend the decomposition via Arnoldi iteration to a size m
7. Go to step 2

Performance test

- Compute the largest 100 eigenvalues
- Basis of 200 vectors
- Tolerance 10^{-7}

Eigenvalue problems from the UF sparse collection

bcsstk18 11948×11948

Andrews 60000×60000

Lin 256000×256000

CPU BLAS

ATLAS 3.8.3 on an Intel Core2 Duo E7300 2.66 GHz

GPU BLAS

CUBLAS 3.0 on a nVidia GeForce GTX 280 (240 cores)

Performance results (simple precision)

	bcsstk18		Andrews		Lin	
	ATLAS	CUBLAS	ATLAS	CUBLAS	ATLAS	CUBLAS
Total	2.393	3.195	33.486	41.456	493.774	634.454
Operator	0.153	0.160	2.485	2.618	16.100	18.473
Extraction	0.033	0.033	0.094	0.089	0.357	0.355
Update	0.152	0.016	2.013	0.161	31.511	2.619
Orthog.	2.053	2.985	28.867	38.586	445.748	612.997
AXPY	0.000	0.005	0.000	0.014	0.004	0.053
SCAL	0.004	0.005	0.043	0.017	0.674	0.110
NRM2	0.015	0.047	0.200	0.151	3.186	0.673
GEMV 'N'	1.195	0.087	16.496	0.810	257.208	11.210
GEMV 'T'	0.837	2.835	12.121	37.580	184.652	600.889

`cublasSgemv` is not optimized for skinny matrices $\Rightarrow$ custom code

Performance results (simple precision)

	bcsstk18		Andrews		Lin	
	ATLAS	CUBLAS	ATLAS	CUBLAS	ATLAS	CUBLAS
Total	2.393	3.195	33.486	41.456	493.774	634.454
Operator	0.153	0.160	2.485	2.618	16.100	18.473
Extraction	0.033	0.033	0.094	0.089	0.357	0.355
Update	0.152	0.016	2.013	0.161	31.511	2.619
Orthog.	2.053	2.985	28.867	38.586	445.748	612.997
AXPY	0.000	0.005	0.000	0.014	0.004	0.053
SCAL	0.004	0.005	0.043	0.017	0.674	0.110
NRM2	0.015	0.047	0.200	0.151	3.186	0.673
GEMV 'N'	1.195	0.087	16.496	0.810	257.208	11.210
GEMV 'T'	0.837	2.835	12.121	37.580	184.652	600.889

`cublasSgemv` is not optimized for skinny matrices $\Rightarrow$ custom code

Performance results (simple precision)

	bcsstk18		Andrews		Lin	
	ATLAS	CUBLAS	ATLAS	CUBLAS	ATLAS	CUBLAS
Total	2.393	3.195	33.486	41.456	493.774	634.454
Operator	0.153	0.160	2.485	2.618	16.100	18.473
Extraction	0.033	0.033	0.094	0.089	0.357	0.355
Update	0.152	0.016	2.013	0.161	31.511	2.619
Orthog.	2.053	2.985	28.867	38.586	445.748	612.997
AXPY	0.000	0.005	0.000	0.014	0.004	0.053
SCAL	0.004	0.005	0.043	0.017	0.674	0.110
NRM2	0.015	0.047	0.200	0.151	3.186	0.673
GEMV 'N'	1.195	0.087	16.496	0.810	257.208	11.210
GEMV 'T'	0.837	2.835	12.121	37.580	184.652	600.889

`cublasSgemv` is not optimized for skinny matrices $\Rightarrow$ custom code

Outline

CUDA parallel model for 1.x devices

Blocks of threads

- All threads share the same code (kernel)
- Up to 8 blocks are scheduled to each processor
- Threads are executed in parallel via warps (32), synchronized at instruction level

Memory hierarchy

Register	private to each thread (64Kb)
Shared	private to each block (16 Kb)
Global	shared among all threads and CPU (> 1 Gb) read only texture cache (8 Kb)
Constant	global, cached and read only (64 Kb)

Rectangular matrix vector product parallelization

GEMV 'T'

$$y = V^* x$$

$$y \in \mathbb{R}^m,\ V \in \mathbb{R}^{n \times m},\ x \in \mathbb{R}^n,\ n \ggg m$$

Multiple vector dot product

- Each block computes $y_i = {v_{1:n,i}}^* x$ (m blocks $i = 1, \ldots, m$)
  - All threads execute the same instructions
- Use a texture to cache x access
- 256 threads per block
  - Maximize processor occupation (hide register latency)
  - Coalesced memory access to $v_{1:n,i}$ (exploit memory bandwidth)
- One reduction operation per block with shared memory (based on Mark Harris code from CUDA SDK)

Kernel code (single precision)

```
#define K 256
texture<float> tex_x;
__global__ void kernel_sgemvt(int l, float alpha, float *v, int ldv,
                              float *x, int offset, float beta, float *y)
{
  __shared__ float sh[K];
  float s = 0.0;
  int tix = threadIdx.x, i, j = blockIdx.x * lda + tix;
  for (i = tix; i < l; i += K, j += K) {
    s += v[j] * tex1Dfetch(tex_x, i + offset);
  }
  sh[tix] = s; __syncthreads();
  if (tix < 128) sh[tix] += shared[tix + 128];
  __syncthreads();
  if (tix <  64) sh[tix] += shared[tix +  64];
  __syncthreads();
  if (tix <  32) {
    sh[tix] += sh[tix + 32]; sh[tix] += sh[tix + 16]; sh[tix] += sh[tix + 8];
    sh[tix] += sh[tix +  4]; sh[tix] += sh[tix +  2]; sh[tix] += sh[tix + 1];
  }
  if (tix == 0) y[blockIdx.x] = alpha * sh[0] + beta * y[blockIdx.x];
}
```

Performance results (single precision)

		ATLAS	CUBLAS	Kernel	Speed-up
bcsstk18	GEMV 'T'	0.837	2.835	0.079	10.5
	Orthog.	2.053	2.985	0.230	8.9
	Total	2.393	3.195	0.437	5.5
Andrews	GEMV 'T'	12.121	37.580	0.570	21.3
	Orthog.	28.867	38.586	1.586	18.2
	Total	33.486	41.456	4.558	7.3
Lin	GEMV 'T'	184.652	600.889	9.152	20.2
	Orthog.	445.748	612.997	21.588	20.6
	Total	493.774	634.454	42.926	11.5

Kernel code (double precision)

```
#define K 256
texture<int2> tex_x;
__global__ void kernel_dgemvt(int l, double alpha, double *v, int ldv,
                              double *x, int offset, double beta, double *y)
{
  __shared__ double shared[K];
  double s = 0.0;
  int tix = threadIdx.x, i, j = blockIdx.x * lda + tix;
  int2 t;
  for (i = tix; i < l; i += K, j += K) {
    t = tex1Dfetch(tex_x, i + offset);
    s += v[j] * __hiloint2double(t.y, t.x);
  }
  sh[tix] = s; __syncthreads();
  if (tix < 128) sh[tix] += shared[tix + 128]; __syncthreads();
  if (tix <  64) sh[tix] += shared[tix +  64]; __syncthreads();
  if (tix <  32) {
    sh[tix] += sh[tix + 32]; sh[tix] += sh[tix + 16]; sh[tix] += sh[tix + 8];
    sh[tix] += sh[tix +  4]; sh[tix] += sh[tix +  2]; sh[tix] += sh[tix + 1];
  }
  if (tix == 0) y[blockIdx.x] = alpha * sh[0] + beta * y[blockIdx.x];
}
```

Performance results (double precision)

		ATLAS	CUBLAS	Kernel	Speed-up
bcsstk18	GEMV 'T'	1.678	4.645	0.134	12.5
	Orthog.	3.590	4.903	0.405	8.9
	Total	4.108	5.151	0.654	6.3
Andrews	GEMV 'T'	22.939	62.344	1.436	16.0
	Orthog.	50.091	63.964	3.133	16.0
	Total	57.018	67.980	7.135	8.0
Lin	GEMV 'T'	383.98	1043.12	23.61	16.3
	Orthog.	843.18	1064.01	45.88	18.4
	Total	931.93	1101.94	83.98	11.1

Although double precision in this graphics processor is 8 times slower than single precision, this computation achieves similar performance because is limited by memory bandwidth

Conclusions and future work

20x speed-up in the Classical Gram-Schmidt procedure

- Krylov subspace is stored on the GPU
- Iterative method runs on the CPU launching CUBLAS kernels
- A custom kernel is the key for performance
- Even for double precision

Future work

- Implement sparse matrix vector product on the GPU
- Join GEMV and NRM2 in one kernel
- Support for complex arithmetic
- Distributed memory parallelization for multiple GPUs
- Port to other architectures (Fermi, OpenCL)
- Extend to other Krylov solvers (GMRES)