Computational Science on Many-Core Architectures

360.252

Karl Rupp

Institute for Microelectronics
Vienna University of Technology
https://www.iue.tuwien.ac.at/

Zoom Channel 95028746244
Wednesday, November 11, 2020

Agenda for Today

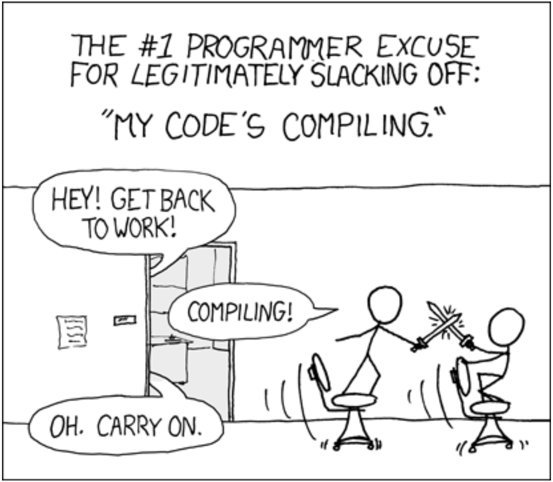
Exercise 3 Recap

Kernel Fusion - Conjugate Gradients

Kernel Fusion - Multiple Dot Products

Exercise 4

Exercise 3 Recap



https://xkcd.com/303/

Exercise 3 Recap

Feedback Time

- How was your experience?

Exercise 3 Recap

Feedback Time

- How was your experience?
- Links to points for course will be sent out today or tomorrow

Exercise 3 Recap

Feedback Time

- How was your experience?
- Links to points for course will be sent out today or tomorrow

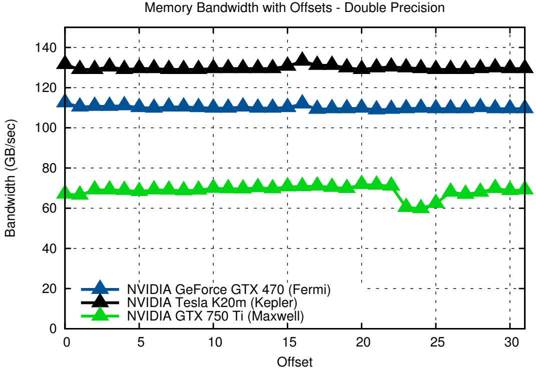
Mind the Temporaries!

```
double dot_product(int N, ...) {
  double result, *cuda_result;
  cudaMalloc(&cuda_result, sizeof(double));
  cuda_dot_product<<<...>>>(...);
  ...
}
void cg() {
  ...
  while (1) {
     ...
     dot_product(...);
  }
}
```

Exercise 3 Recap

Offset Memory Access

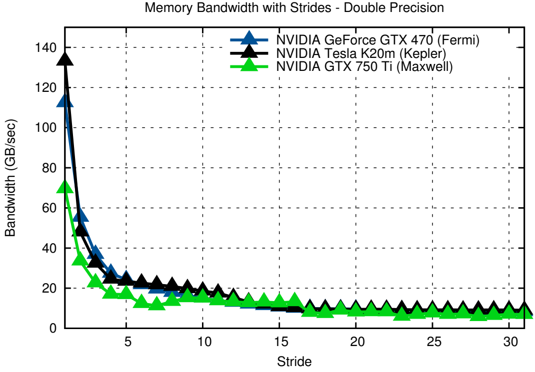
```
__global__
void work(double *x, double *y, double *z, int N, int k)
{
  int thread_id = blockIdx.x*blockDim.x + threadIdx.x;
  for (size_t i=thread_id; i<N; i += blockDim.x * gridDim.x)
    z[i+k] = x[i+k] + y[i+k];
}
```



Exercise 3 Recap

Strided Memory Access

```
__global__
void work(double *x, double *y, double *z, int N, int k)
{
  int thread_id = blockIdx.x*blockDim.x + threadIdx.x;
  for (size_t i=thread_id; i<N; i += blockDim.x * gridDim.x)
    z[i*k] = x[i*k] + y[i*k];
}
```



Exercise 3 Recap

Strided Memory Access

- Array of structs problematic

```
typedef struct particle
{
  double pos_x; double pos_y; double pos_z;
  double vel_x; double vel_y; double vel_z;
  double mass;
} Particle;

__global__
void increase_mass(Particle *particles, int N)
{
  int thread_id = blockIdx.x*blockDim.x + threadIdx.x;
  for (int i=thread_id; i<N; i += blockDim.x * gridDim.x)
    particles[i].mass *= 2.0;
}
```

Exercise 3 Recap

Strided Memory Access

- Workaround: Structure of Arrays

```
typedef struct particles
{
  double *pos_x; double *pos_y; double *pos_z;
  double *vel_x; double *vel_y; double *vel_z;
  double *mass;
} Particle;

__global__
void increase_mass(Particle *particles, int N)
{
  int thread_id = blockIdx.x*blockDim.x + threadIdx.x;
  for (int i=thread_id; i<N; i += blockDim.x * gridDim.x)
    particles.mass[i] *= 2.0;
}
```

Conjugate Gradients

Pseudocode

Choose x_0
$p_0 = r_0 = b - Ax_0$
For $i = 0$ until convergence

1. Compute and store Ap_i
2. Compute $\langle p_i, Ap_i \rangle$
3. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
4. $x_{i+1} = x_i + \alpha_i p_i$
5. $r_{i+1} = r_i - \alpha_i Ap_i$
6. Compute $\langle r_{i+1}, r_{i+1} \rangle$
7. $\beta_i = \langle r_{i+1}, r_{i+1} \rangle / \langle r_i, r_i \rangle$
8. $p_{i+1} = r_{i+1} + \beta_i p_i$

EndFor

BLAS-based Implementation

-
SpMV, AXPY
For $i = 0$ until convergence

1. SpMV
2. DOT
3. -
4. AXPY
5. AXPY
6. DOT
7. -
8. AXPY

EndFor

Conjugate Gradients

Pseudocode

Choose x_0
$p_0 = r_0 = b - Ax_0$
For $i = 0$ until convergence

1. Compute and store Ap_i
2. Compute $\langle p_i, Ap_i \rangle$
3. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
4. $x_{i+1} = x_i + \alpha_i p_i$
5. $r_{i+1} = r_i - \alpha_i Ap_i$
6. Compute $\langle r_{i+1}, r_{i+1} \rangle$
7. $\beta_i = \langle r_{i+1}, r_{i+1} \rangle / \langle r_i, r_i \rangle$
8. $p_{i+1} = r_{i+1} + \beta_i p_i$

EndFor

BLAS-based Implementation

-
SpMV, AXPY
For $i = 0$ until convergence

1. SpMV
2. DOT $\leftarrow$ Global sync!
3. -
4. AXPY
5. AXPY
6. DOT $\leftarrow$ Global sync!
7. -
8. AXPY

EndFor

Conjugate Gradients

Pseudocode

Choose x_0
$p_0 = r_0 = b - Ax_0$
For $i = 0$ until convergence

1. Compute and store Ap_i
2. Compute $\langle p_i, Ap_i \rangle$
3. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
4. $x_{i+1} = x_i + \alpha_i p_i$
5. $r_{i+1} = r_i - \alpha_i Ap_i$
6. Compute $\langle r_{i+1}, r_{i+1} \rangle$
7. $\beta_i = \langle r_{i+1}, r_{i+1} \rangle / \langle r_i, r_i \rangle$
8. $p_{i+1} = r_{i+1} + \beta_i p_i$

EndFor

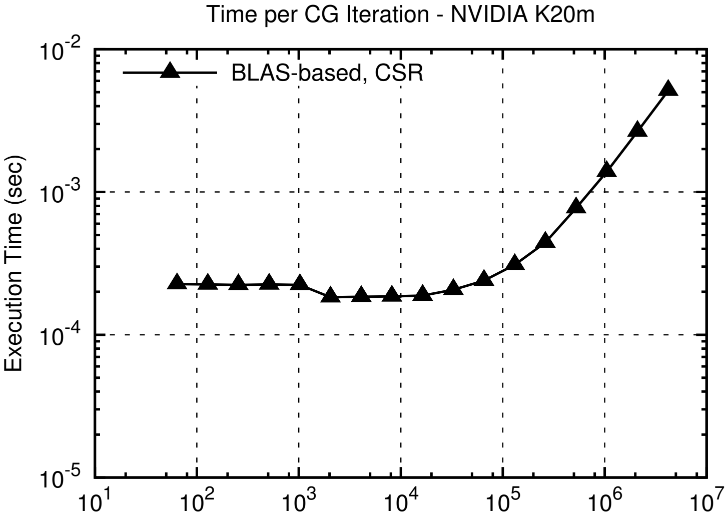
BLAS-based Implementation

-
SpMV, AXPY
For $i = 0$ until convergence

1. SpMV $\leftarrow$ No caching of Ap_i
2. DOT $\leftarrow$ Global sync!
3. -
4. AXPY
5. AXPY $\leftarrow$ No caching of r_{i+1}
6. DOT $\leftarrow$ Global sync!
7. -
8. AXPY

EndFor

Conjugate Gradients



- (2D Finite Difference Discretization)

Conjugate Gradient

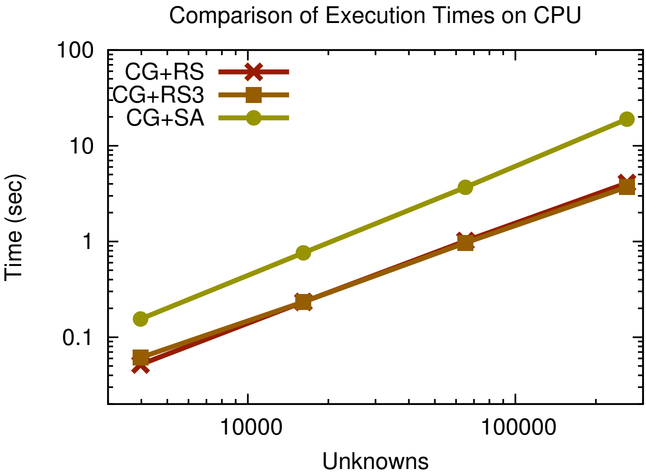
Implications

- Kernel launches expensive
- Delicate balance for preconditioners

Conjugate Gradient

Implications

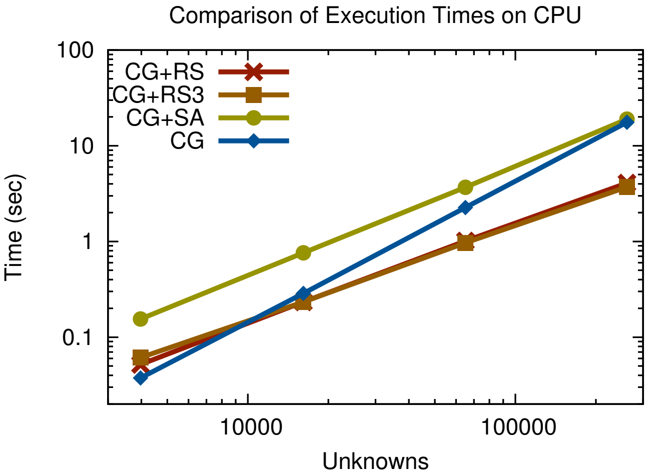
- Kernel launches expensive
- Delicate balance for preconditioners



Conjugate Gradient

Implications

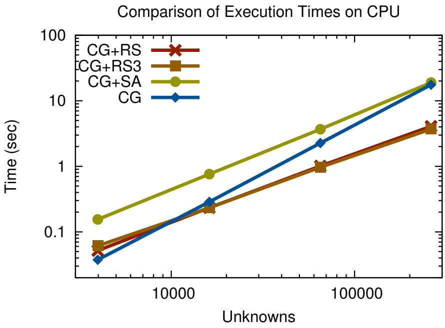
- Kernel launches expensive
- Delicate balance for preconditioners



Conjugate Gradient

Implications

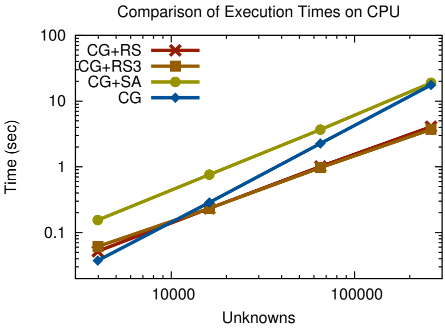
- Kernel launches expensive
- Delicate balance for preconditioners

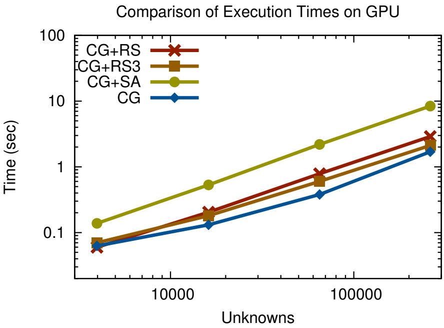


Conjugate Gradient

Implications

- Kernel launches expensive
- Delicate balance for preconditioners





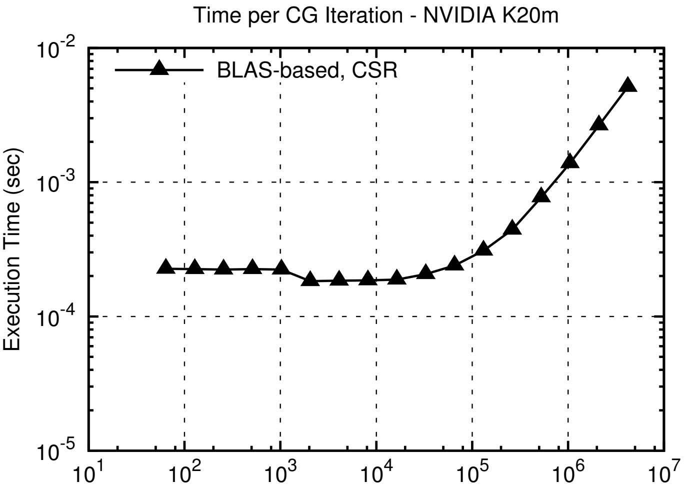
Conjugate Gradient Optimizations

Optimization 1

- Get best performance out of SpMV
- Compare different sparse matrix types

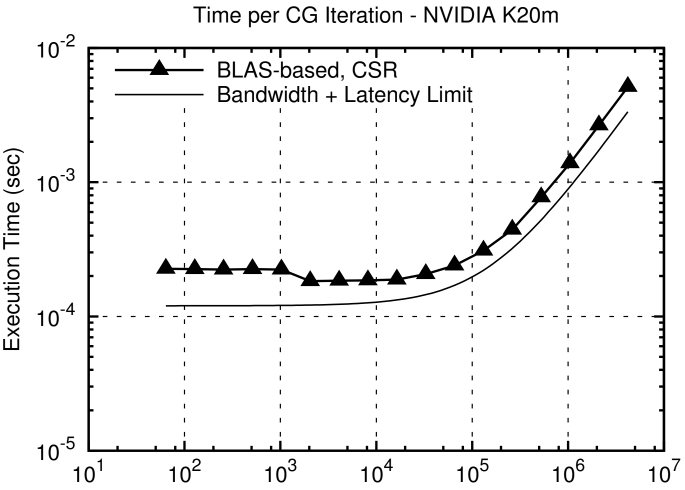
Cf.: N. Bell: Implementing sparse matrix-vector multiplication
on throughput-oriented processors. *Proc. SC '09*

Conjugate Gradients



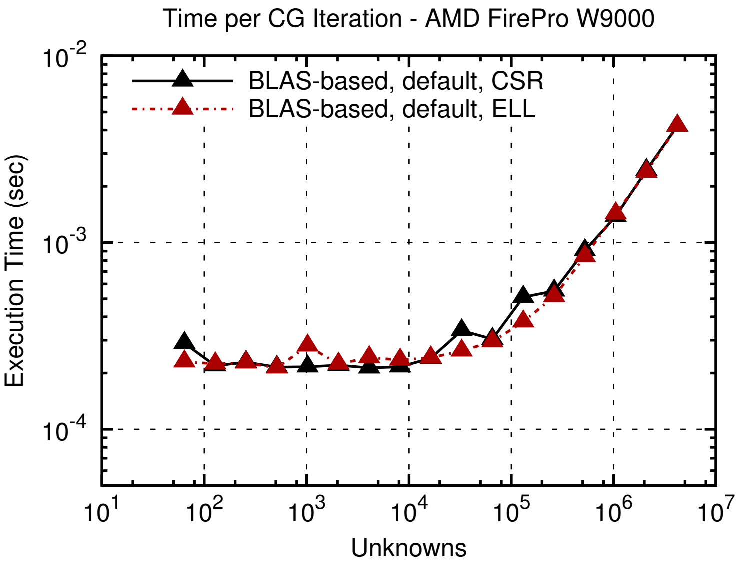
- (2D Finite Difference Discretization)

Conjugate Gradients



- (2D Finite Difference Discretization)

Conjugate Gradients



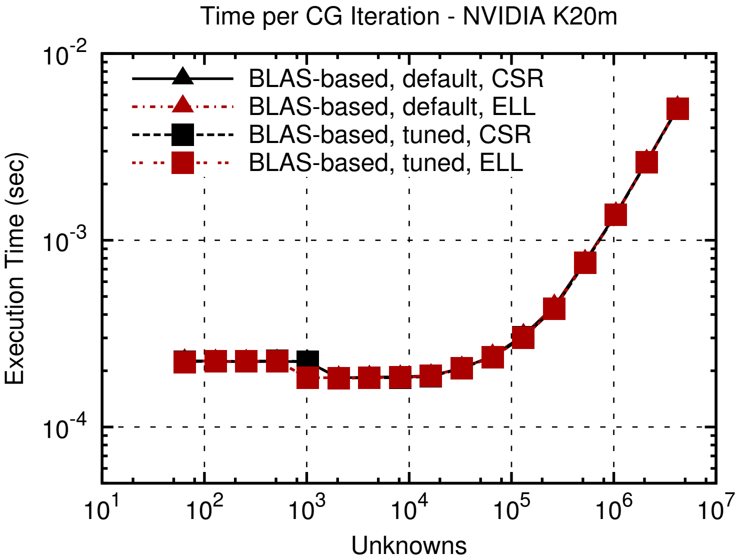
- (2D Finite Difference Discretization)

Conjugate Gradient Optimizations

Optimization 2

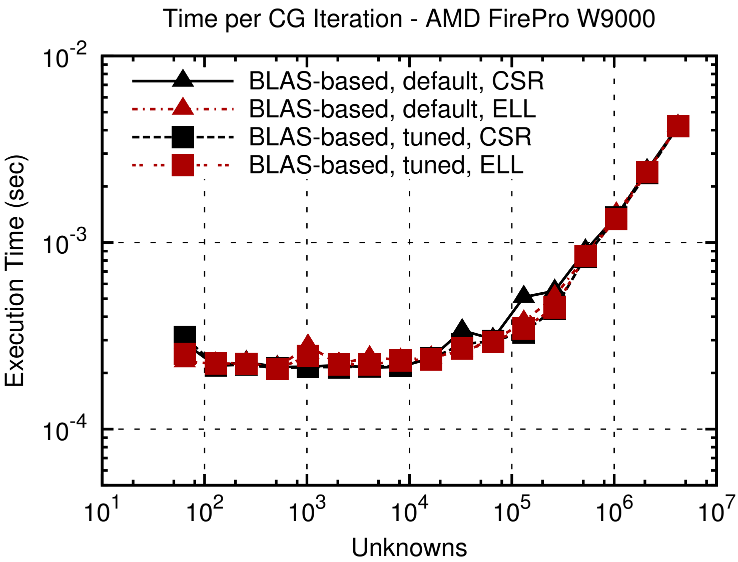
- Optimize kernel parameters for each operation

Conjugate Gradients



- (2D Finite Difference Discretization)

Conjugate Gradients



- (2D Finite Difference Discretization)

Conjugate Gradient Optimizations

Optimization 3: Rearrange the algorithm

- Remove unnecessary reads
- Remove unnecessary synchronizations
- Use custom kernels instead of standard BLAS

Conjugate Gradients

Standard CG

Choose x_0
$p_0 = r_0 = b - Ax_0$
For $i = 0$ until convergence

1. Compute and store Ap_i
2. Compute $\langle p_i, Ap_i \rangle$
3. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
4. $x_{i+1} = x_i + \alpha_i p_i$
5. $r_{i+1} = r_i - \alpha_i Ap_i$
6. Compute $\langle r_{i+1}, r_{i+1} \rangle$
7. $\beta_i = \langle r_{i+1}, r_{i+1} \rangle / \langle r_i, r_i \rangle$
8. $p_{i+1} = r_{i+1} + \beta_i p_i$

EndFor

Conjugate Gradients

Standard CG

Choose x_0
$p_0 = r_0 = b - Ax_0$
For $i = 0$ until convergence

1. Compute and store Ap_i
2. Compute $\langle p_i, Ap_i \rangle$
3. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
4. $x_{i+1} = x_i + \alpha_i p_i$
5. $r_{i+1} = r_i - \alpha_i Ap_i$
6. Compute $\langle r_{i+1}, r_{i+1} \rangle$
7. $\beta_i = \langle r_{i+1}, r_{i+1} \rangle / \langle r_i, r_i \rangle$
8. $p_{i+1} = r_{i+1} + \beta_i p_i$

EndFor

Pipelined CG

Choose x_0
$p_0 = r_0 = b - Ax_0$
For $i = 1$ until convergence

1. $i = 1$: Compute α_0, β_0, Ap_0
2. $x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$
3. $r_i = r_{i-1} - \alpha_{i-1} Ap_i$
4. $p_i = r_i + \beta_{i-1} p_{i-1}$
5. Compute and store Ap_i
6. Compute $\langle Ap_i, Ap_i \rangle$, $\langle p_i, Ap_i \rangle$, $\langle r_i, r_i \rangle$
7. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
8. $\beta_i = (\alpha_i^2 \langle Ap_i, Ap_i \rangle - \langle r_i, r_i \rangle) / \langle r_i, r_i \rangle$

EndFor

Conjugate Gradients

Standard CG

Choose x_0
$p_0 = r_0 = b - Ax_0$
For $i = 0$ until convergence

1. Compute and store Ap_i
2. Compute $\langle p_i, Ap_i \rangle$
3. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
4. $x_{i+1} = x_i + \alpha_i p_i$
5. $r_{i+1} = r_i - \alpha_i Ap_i$
6. Compute $\langle r_{i+1}, r_{i+1} \rangle$
7. $\beta_i = \langle r_{i+1}, r_{i+1} \rangle / \langle r_i, r_i \rangle$
8. $p_{i+1} = r_{i+1} + \beta_i p_i$

EndFor

Pipelined CG

Choose x_0
$p_0 = r_0 = b - Ax_0$
For $i = 1$ until convergence

1. $i = 1$: Compute α_0, β_0, Ap_0
2. $x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$
3. $r_i = r_{i-1} - \alpha_{i-1} Ap_i$
4. $p_i = r_i + \beta_{i-1} p_{i-1}$
5. Compute and store Ap_i
6. Compute $\langle Ap_i, Ap_i \rangle$, $\langle p_i, Ap_i \rangle$, $\langle r_i, r_i \rangle$
7. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
8. $\beta_i = (\alpha_i^2 \langle Ap_i, Ap_i \rangle - \langle r_i, r_i \rangle) / \langle r_i, r_i \rangle$

EndFor

Conjugate Gradients

Standard CG

Choose x_0
$p_0 = r_0 = b - Ax_0$
For $i = 0$ until convergence

1. Compute and store Ap_i
2. Compute $\langle p_i, Ap_i \rangle$
3. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
4. $x_{i+1} = x_i + \alpha_i p_i$
5. $r_{i+1} = r_i - \alpha_i Ap_i$
6. Compute $\langle r_{i+1}, r_{i+1} \rangle$
7. $\beta_i = \langle r_{i+1}, r_{i+1} \rangle / \langle r_i, r_i \rangle$
8. $p_{i+1} = r_{i+1} + \beta_i p_i$

EndFor

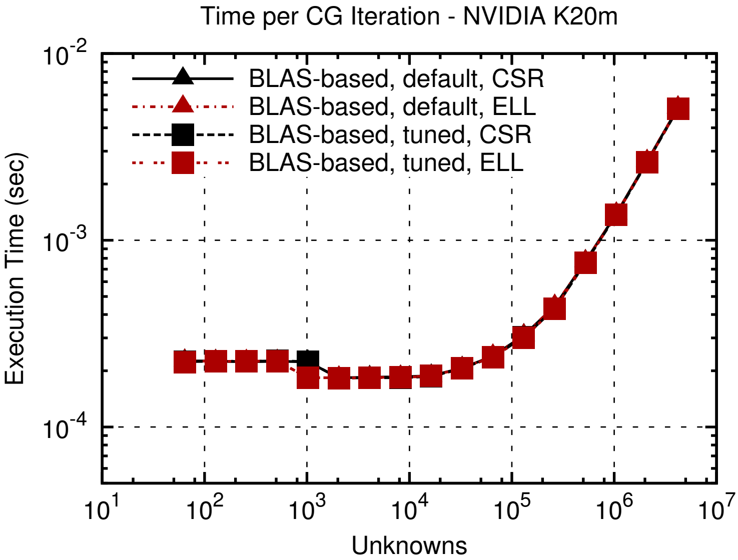
Pipelined CG

Choose x_0
$p_0 = r_0 = b - Ax_0$
For $i = 1$ until convergence

1. $i = 1$: Compute α_0, β_0, Ap_0
2. $x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$
3. $r_i = r_{i-1} - \alpha_{i-1} Ap_i$
4. $p_i = r_i + \beta_{i-1} p_{i-1}$
5. Compute and store Ap_i
6. Compute $\langle Ap_i, Ap_i \rangle$, $\langle p_i, Ap_i \rangle$, $\langle r_i, r_i \rangle$
7. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
8. $\beta_i = (\alpha_i^2 \langle Ap_i, Ap_i \rangle - \langle r_i, r_i \rangle) / \langle r_i, r_i \rangle$

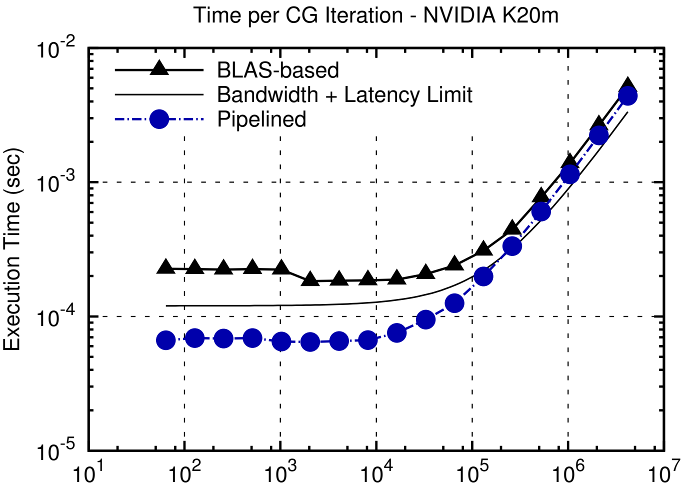
EndFor

Conjugate Gradients



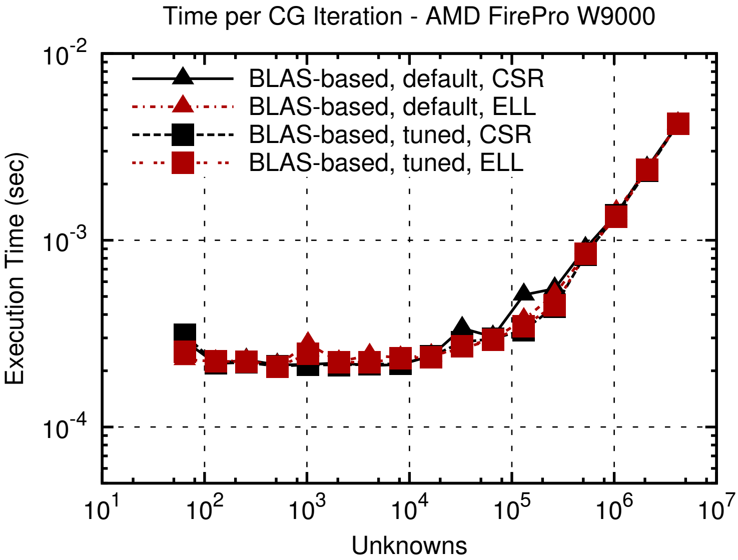
- (2D Finite Difference Discretization)

Conjugate Gradients



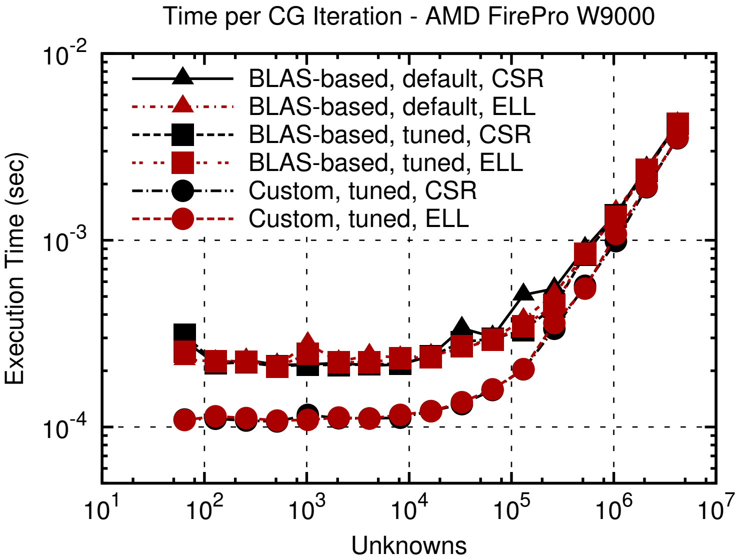
- (2D Finite Difference Discretization)

Conjugate Gradients



- (2D Finite Difference Discretization)

Conjugate Gradients



- (2D Finite Difference Discretization)

Kernel Fusion - Multiple Dot Products

Gram-Schmidt method

- Given orthonormal basis $\{v_1, v_2, \ldots, v_N\}$, augment by w
- $w \leftarrow w - \langle w, v_i \rangle v_i$
- $w \leftarrow w / \|w\|$
- Add w to basis

Multiple inner products $\langle w, v_i \rangle$

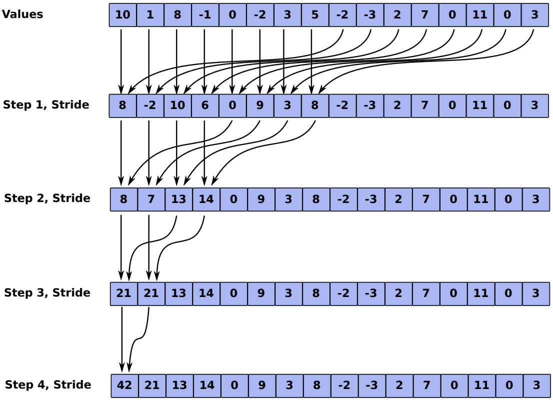
- Performance critical (global reductions)
- Reuse of w desirable

Implementation 1: dot

Goal: Compute $\alpha_i = \langle w, v_i \rangle$ for $i = 1, \ldots, N$ efficiently

Method 1: Iterated application of *dot()*

- Decompose into N separate inner products
- N calls to BLAS level 1 routine $ddot$
- Reductions: One per work group, one over results of work groups



Implementation 2: gemv

Goal: Compute $\alpha_i = \langle w, v_i \rangle$ for $i = 1, \ldots, N$ efficiently

Method 2: Pack vectors into matrix, use *gemv*

- Set $\mathbf{A} = \begin{pmatrix} v_1^{\mathrm{T}} \\ \vdots \\ v_N^{\mathrm{T}} \end{pmatrix} \in \mathbb{R}^{N \times M}, N \ll M$
- Compute $\boldsymbol{\alpha} = \mathbf{A}\mathbf{x}$
- One BLAS level 2 *dgemv* call

Implementation 3: mdot

Method 3: Custom routine *mdot*

- Process $\alpha_i = \langle w, v_i \rangle$ in batches
- Batch sizes 1, 2, 3, 4, 8
- Load entries of w only once per batch
- Fit remaining inner products into largest batch possible
- Batch size 8: Only 12.5% overhead vs. arbitrary batch sizes

Implementation 3: mdot

Method 3: Custom routine *mdot*

- Process $\alpha_i = \langle w, v_i \rangle$ in batches
- Batch sizes 1, 2, 3, 4, 8
- Load entries of w only once per batch
- Fit remaining inner products into largest batch possible
- Batch size 8: Only 12.5% overhead vs. arbitrary batch sizes

Code sketch (Batch size 4)

```
for (size_t i=thread_id; i<M; i += threads_per_group)
{
  double val_w = w[i];
  alpha_1 += val_w * v1[i];
  alpha_2 += val_w * v2[i];
  alpha_3 += val_w * v3[i];
  alpha_4 += val_w * v4[i];
}
```

Benchmarks

Part 3: Benchmarks

Exercises

Environment

- `https://gtx1080.360252.org/2020/ex4/`
- (Might receive visual updates and additional hints over the next days)
- Due: Tuesday, November 17, 2020 at 23:59pm

Hints and Suggestions

- Consider version control for locally developed code
- Please let me know of any bugs or issues
- Example codes and code skeletons provided at URL above