Efficient Serial and Parallel Coordinate Descent Methods for Huge-Scale Convex Optimization

Martin Takáč

The University of Edinburgh

Based on:


October 06, 2011
Outline

1. Randomized Coordinate Descent Method
   - Theory
   - GPU implementation
   - Numerical examples
2. Facing the Multicore-Challenge II (a few insights from a conference)
   - StarSs
   - Intel Cilk Plus
   - SIMD and auto-vectorization
3. Discussion
Randomized Coordinate Descent Method
Problem Formulation

\[ \min_{x \in \mathbb{R}^N} F(x) \overset{\text{def}}{=} f(x) + \Psi(x), \quad (P) \]

where

- \( f \) is convex and smooth
- \( \Psi \) is convex, simple and block-separable
- \( N \) is huge (\( N \gg 10^6 \))

<table>
<thead>
<tr>
<th>( \Psi(x) )</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>smooth minimization</td>
</tr>
<tr>
<td>( \lambda |x|_1 )</td>
<td>sparsity-inducing term</td>
</tr>
<tr>
<td>indicator function of a set</td>
<td>block-separable constraints</td>
</tr>
</tbody>
</table>
Problem Formulation

\[
\min_{x \in \mathbb{R}^N} F(x) \overset{\text{def}}{=} f(x) + \Psi(x), \quad (P)
\]

where

- \( f \) is convex and smooth
- \( \Psi \) is convex, simple and block-separable
- \( N \) is huge \((N \gg 10^6)\)

<table>
<thead>
<tr>
<th>( \Psi(x) )</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>smooth minimization</td>
</tr>
<tr>
<td>( \lambda |x|_1 )</td>
<td>sparsity-inducing term</td>
</tr>
<tr>
<td>indicator function of a set</td>
<td>block-separable constraints</td>
</tr>
</tbody>
</table>

General Iterative Algorithm

1. choose initial point \( x := x_0 \in \mathbb{R}^n \)
2. iterate
   - choose direction \( d := d(x) \)
   - choose step length \( \alpha(d, x) \)
   - set \( x = x + \alpha d \)
Problem Formulation

\[
\min_{x \in \mathbb{R}^N} F(x) \overset{\text{def}}{=} f(x) + \Psi(x), \quad (P)
\]

where

- \(f\) is convex and smooth
- \(\Psi\) is convex, simple and block-separable
- \(N\) is huge (\(N \gg 10^6\))

<table>
<thead>
<tr>
<th>(\Psi(x))</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>smooth minimization</td>
</tr>
<tr>
<td>(\lambda |x|_1)</td>
<td>sparsity-inducing term</td>
</tr>
<tr>
<td>indicator function of a set</td>
<td>block-separable constraints</td>
</tr>
</tbody>
</table>

General Iterative Algorithm

1. choose initial point \(x := x_0 \in \mathbb{R}^n\)
2. iterate
   - choose direction \(d := d(x)\)
   - choose step length \(\alpha(d, x)\)
   - set \(x = x + \alpha d\)

- More sophisticated direction \(\Rightarrow\) less iterations
- More sophisticated direction \(\Rightarrow\) more work per iteration
Why Coordinate Descent?

- a very old optimization method (B. O'Donnell 1967)
- can move only in directions \{e_1, e_2, \ldots, e_N\}, where
  \[ e_i = (0, 0, 0, \ldots, 0, 1, 0, \ldots, 0)^T \]
Why Coordinate Descent?

- a very old optimization method (B. O'Donnell 1967)
- can move only in directions \( \{e_1, e_2, \ldots, e_N\} \), where 
  \[ e_i = (0, 0, 0, \ldots, 0, 1, 0, \ldots, 0)^T \]

**Methods for choosing direction**

- cyclic (or essentially cyclic)
- coordinate with the maximal directional derivative (**Greedy**)
- **RANDOM** (**WHY?**)
Why Coordinate Descent?

- a very old optimization method (B. O'Donnell 1967)
- can move only in directions \( \{e_1, e_2, \ldots, e_N\} \), where \\
  \[ e_i = (0, 0, 0, \ldots, 0, 1, 0, \ldots, 0)^T \]

Methods for choosing direction

- cyclic (or essentially cyclic)
- coordinate with the maximal directional derivative (Greedy)
- RANDOM (WHY?)

Applications

- Sparse regression, LASSO
- Truss topology design
- (Sparse) group LASSO
- Elastic Net . . .
- \( L_1 \) regularized linear support vector machines (SVM) with various (smooth) loss functions
Randomized Coordinate Descent: a 2D example

\[ f(x) = \frac{1}{2}x^T \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix} x - (1.5 \ 1.5) x, \quad x_0 = (0 \ 0)^T \]
Randomized Coordinate Descent: a 2D example

\[ f(x) = \frac{1}{2} x^T \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix} x - \begin{pmatrix} 1.5 \\ 1.5 \end{pmatrix} x, \quad x_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}^T \]

What we need?
- Generate random coordinate (we need \( n \) dimensional coin)
Randomized Coordinate Descent: a 2D example

\[ f(x) = \frac{1}{2}x^T \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix} x - (1.5 \ 1.5) x, \quad x_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}^T \]

What we need?

- Generate random coordinate (we need \( n \) dimensional coin)
- Luckily in our case \( n = 2 \)
Randomized Coordinate Descent: a 2D example

\[ f(x) = \frac{1}{2} x^T \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix} x - \begin{pmatrix} 1.5 \\ 1.5 \end{pmatrix} x, \quad x_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}^T \]
Randomized Coordinate Descent: a 2D example

\[ f(x) = \frac{1}{2} x^T \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix} x - \begin{pmatrix} 1.5 \\ 1.5 \end{pmatrix} x, \quad x_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}^T \]
Randomized Coordinate Descent: a 2D example

\[ f(x) = \frac{1}{2} x^T \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix} x - \begin{pmatrix} 1.5 \\ 1.5 \end{pmatrix} x, \quad x_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}^T \]
Randomized Coordinate Descent: a 2D example

\[ f(x) = \frac{1}{2} x^T \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix} x - \begin{pmatrix} 1.5 \\ 1.5 \end{pmatrix} x, \quad x_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}^T \]
Randomized Coordinate Descent: a 2D example

\[ f(x) = \frac{1}{2} x^T \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix} x - \begin{pmatrix} 1.5 \\ 1.5 \end{pmatrix} x, \quad x_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}^T \]
Randomized Coordinate Descent: a 2D example

\[ f(x) = \frac{1}{2} x^T \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix} x - \begin{pmatrix} 1.5 & 1.5 \end{pmatrix} x, \quad x_0 = \begin{pmatrix} 0 & 0 \end{pmatrix}^T \]
Randomized Coordinate Descent: a 2D example

\[ f(x) = \frac{1}{2} x^T \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix} x - \begin{pmatrix} 1.5 \\ 1.5 \end{pmatrix} x, \quad x_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}^T \]
Randomized Coordinate Descent: a 2D example

\[ f(x) = \frac{1}{2} x^T \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix} x - \begin{pmatrix} 1.5 & 1.5 \end{pmatrix} x, \quad x_0 = \begin{pmatrix} 0 & 0 \end{pmatrix}^T \]
Randomized Coordinate Descent: a 2D example

\[ f(x) = \frac{1}{2} x^T \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix} x - \begin{pmatrix} 1.5 \\ 1.5 \end{pmatrix} x, \quad x_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}^T \]
Any vector $x \in \mathbb{R}^N$ can be written uniquely as

$$x = \sum_{i=1}^{n} U_i x^{(i)}, \quad \text{where} \quad x^{(i)} = U_i^T x \in \mathbb{R}_i \equiv \mathbb{R}^{N_i}.$$
Block norms (norms in $\mathbb{R}_i$): We equip $\mathbb{R}_i$ with a pair of conjugate Euclidean norms:

$$\|t\|_{(i)} = \langle B_i t, t \rangle^{1/2}, \quad \|t\|_{(i)}^* = \langle B_i^{-1} t, t \rangle^{1/2}, \quad t \in \mathbb{R}_i,$$

where $B_i \in \mathbb{R}^{N_i \times N_i}$ is a positive definite matrix.

Induced norms (norms in $\mathbb{R}^N$):

For fixed positive scalars $w_1, \ldots, w_n$ and $x \in \mathbb{R}^N$ we define

$$\|x\|_W = \left[ \sum_{i=1}^n w_i \|x^{(i)}\|_{(i)}^2 \right]^{1/2}.$$

Conjugate norm is defined by

$$\|y\|_W^* = \max_{\|x\|_W \leq 1} \langle y, x \rangle = \left[ \sum_{i=1}^n w_i^{-1}(\|y^{(i)}\|_{(i)}^*)^2 \right]^{1/2},$$

where $W = \text{Diag}(w_1, \ldots, w_n)$. 
Assumptions on $f$ and $\Psi$

The gradient of $f$ is block coordinate-wise Lipschitz with positive constants $L_1, \ldots, L_n$:

$$\|\nabla_i f(x + U_i t) - \nabla_i f(x)\|_{(i)}^* \leq L_i \|t\|_{(i)}, \ x \in \mathbb{R}^N, \ t \in \mathbb{R}, \ i = 1, \ldots, n,$$

where

$$\nabla_i f(x) \overset{\text{def}}{=} (\nabla f(x))^{(i)} = U_i^T \nabla f(x) \in \mathbb{R}_i.$$

An important consequence of (1) is the following inequality

$$f(x + U_i t) \leq f(x) + \langle \nabla_i f(x), t \rangle + \frac{L_i}{2} \|t\|_{(i)}^2. \quad (2)$$

We denote $L = \text{Diag}(L_1, \ldots, L_n)$.

$\Psi$ is block-separable:

$$\Psi(x) = \sum_{i=1}^{n} \Psi_i(x^{(i)}).$$
An upper bound on $F(x + U_i t)$ is readily available:

$$F(x + U_i t) = f(x + U_i t) + \Psi(x + U_i t)$$

\[(2)\]

$$\leq f(x) + \langle \nabla i f(x), t \rangle + \frac{L_i}{2} \| t \|_{(i)}^2 + \Psi_i(x^{(i)} + t) + \sum_{j \neq i} \Psi_j(x^{(j)}).$$

\[\text{def} V_i(x, t)\]

Algorithm 1 UCDC($x_0$)

```
for $k = 0, 1, 2, \ldots$
    Choose block coordinate $i \in \{1, 2, \ldots, n\}$ with probability $\frac{1}{n}$
    $T^{(i)}(x_k) = \text{arg min}\{ V_i(x_k, t) : t \in \mathbb{R}_i \equiv \mathbb{R}^N_i \}$
    $x_{k+1} = x_k + U_i T^{(i)}(x_k)$
end for
```

- In each step we minimize a function of $N_i$ variables
- We assume $\Psi$ is simple: $T^{(i)}(x_k)$ can be computed in closed form
- If $\Psi \equiv 0$, then $T^{(i)}(x_k) = -\frac{1}{L_i} B_i^{-1} \nabla i f(x)$
The Main Result

Theorem 1

Choose initial point \( x_0 \) and confidence level \( \rho \in (0, 1) \). Further, let the target accuracy \( \epsilon > 0 \) and iteration counter \( k \) be chosen in any of the following two ways:

(i) \( \epsilon < F(x_0) - F^* \) and

\[
k \geq \frac{2n \max \{ R^2_L(x_0), F(x_0) - F^* \} }{\epsilon} \left( 1 + \log \frac{1}{\rho} \right) + 2 - \frac{2n \max \{ R^2_L(x_0), F(x_0) - F^* \} }{ F(x_0) - F^* } ,
\]

(ii) \( \epsilon < \min \{ R^2_L(x_0), F(x_0) - F^* \} \) and

\[
k \geq \frac{2n R^2_L(x_0) }{\epsilon} \log \frac{ F(x_0) - F^* }{ \epsilon \rho } .
\]

If \( x_k \) is the random point generated by \( \text{UCDC}(x_0) \) as applied to \( F \), then

\[
P( F(x_k) - F^* \leq \epsilon ) \geq 1 - \rho .
\]

\( R^2_W(x_0) = \max_x \{ \max_{x^* \in X^*} \| x - x^* \|_W^2 : F(x) \leq F(x_0) \} \)
Theorem 2

Let $F$ be strongly convex with respect to $\| \cdot \|_L$ with convexity parameter $\mu > 0$ and choose accuracy level $\epsilon > 0$, confidence level $0 < \rho < 1$, and

$$k \geq \frac{n}{1 - \gamma_\mu} \log \left( \frac{F(x_0) - F^*}{\rho \epsilon} \right),$$

where $\gamma_\mu$ is given by

$$\gamma_\mu = \begin{cases} 1 - \frac{\mu}{4}, & \text{if } \mu \leq 2, \\ \frac{1}{\mu}, & \text{otherwise.} \end{cases} \quad (3)$$

If $x_k$ is the random point generated by UCDC($x_0$), then

$$\mathbb{P}(F(x_k) - F^* \leq \epsilon) \geq 1 - \rho.$$
Comparison with other coordinate descent methods for composite minimization

<table>
<thead>
<tr>
<th></th>
<th>Lip</th>
<th>$\Psi$</th>
<th>block</th>
<th>coordinate choice</th>
<th>$\mathbb{E}(F(x_k) &lt; \epsilon)$</th>
<th>1 iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>[YT09]</td>
<td>$L(\nabla f)$</td>
<td>separable</td>
<td>Yes</td>
<td>greedy</td>
<td>$O\left(\frac{nL(\nabla f)|x^*-x_0|_2^2}{\epsilon}\right)$</td>
<td>$O(n^2)$ cheap</td>
</tr>
<tr>
<td>[SST09]</td>
<td>$L_i = \beta$</td>
<td>$|\cdot|_1$</td>
<td>No</td>
<td>$\frac{1}{n}$</td>
<td>$O\left(\frac{n\beta|x^*-x_0|_2^2}{\epsilon}\right)$</td>
<td>cheap</td>
</tr>
<tr>
<td>[ST10]</td>
<td>$L(\nabla f)$</td>
<td>$|\cdot|_1$</td>
<td>No</td>
<td>cyclic</td>
<td>$O\left(\frac{nL(\nabla f)|x^*-x_0|_2^2}{\epsilon}\right)$</td>
<td>cheap</td>
</tr>
<tr>
<td>Alg 1</td>
<td>$L_i$</td>
<td>separable</td>
<td>Yes</td>
<td>$\frac{1}{n}$</td>
<td>$O\left(\frac{n|x^*-x_0|_2^2}{\epsilon}\right)$</td>
<td>cheap</td>
</tr>
</tbody>
</table>


Comparison with Nesterov’s randomized coordinate descent

... for achieving $P(F(x_k) - F^* \leq \epsilon) \geq 1 - \rho$ in the general case ($F$ not strongly convex):

<table>
<thead>
<tr>
<th>Alg</th>
<th>$\Psi$</th>
<th>$p_i$</th>
<th>norms</th>
<th>complexity</th>
<th>objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>[N10]</td>
<td>0</td>
<td>$\frac{1}{n}$</td>
<td>Euclid.</td>
<td>$\frac{8nR^2_L(x_0)}{\epsilon} \log \frac{4(f(x_0) - f^*)}{\epsilon \rho}$</td>
<td>$f(x) + \frac{\epsilon |x-x_0|^2_L}{8R^2_L(x_0)}$</td>
</tr>
<tr>
<td>[N10]</td>
<td>0</td>
<td>$\frac{L_i}{S}$</td>
<td>Euclid.</td>
<td>$(2n + \frac{8S R^2_i(x_0)}{\epsilon}) \log \frac{4(f(x_0) - f^*)}{\epsilon \rho}$</td>
<td>$f(x) + \frac{\epsilon |x-x_0|^2_i}{8R^2_i(x_0)}$</td>
</tr>
<tr>
<td>Alg1</td>
<td>0</td>
<td>$&gt; 0$</td>
<td>general</td>
<td>$\frac{2R^2_{LP-1}(x_0)}{\epsilon} (1 + \log \frac{1}{\rho}) - 2$</td>
<td>$f$</td>
</tr>
<tr>
<td>Alg2</td>
<td>$\neq 0$</td>
<td>$\frac{1}{n}$</td>
<td>Euclid.</td>
<td>$\frac{2nM}{\epsilon} (1 + \log \frac{1}{\rho})$</td>
<td>$F$</td>
</tr>
</tbody>
</table>

Symbols used in the table: $S = \sum L_i$, $M = \max\{R^2_L(x_0), F(x_0) - F^\}$.

Application: Sparse Regression

Problem formulation & Lipschitz constants

\[
\min_{x \in \mathbb{R}^N} \frac{1}{2} \|Ax - b\|_2^2 + \gamma \|x\|_1
\]

\[L_i = a_i^T a_i\] (easy to compute)

Note: Lipschitz constant of \(\nabla f = \lambda_{\text{max}}(A^T A)\) (hard to compute)

Algorithm 1 UCDC(\(x_0\))

for \(k = 0, 1, 2, \ldots\)

Choose block coor. \(i\)

Compute \(T^{(i)}(x_k)\)

\[x_{k+1} = x_k + U_i T^{(i)}(x_k)\]

end for

Choose \(x_0 = 0\) and set \(g_0 = Ax_0 - b = -b\)

for \(k = 0, 1, 2, \ldots\) do

Choose \(i_k = i \in \{1, 2, \ldots, N\}\) uniformly

\[\nabla_i f(x_k) = a_i^T g_k,\]

\[T^{(i)}(x_k) = \begin{cases} 
- \frac{\nabla_i f(x_k) + \gamma}{L_i}, & \text{if } x_k^{(i)} - \frac{\nabla_i f(x_k) + \gamma}{L_i} > 0, \\
- \frac{\nabla_i f(x_k) - \gamma}{L_i}, & \text{if } x_k^{(i)} - \frac{\nabla_i f(x_k) - \gamma}{L_i} < 0, \\
-x_k^{(i)}, & \text{otherwise,}
\end{cases}\]

\[x_{k+1} = x_k + e_i T^{(i)}(x_k),\]

\[g_{k+1} = g_k + a_i T^{(i)}(x_k).\]

end for
Instance 1 run on CPU: $A \in \mathbb{R}^{10^7 \times 10^6}$, $\|A\|_0 = 10^8$

<table>
<thead>
<tr>
<th>$k/N$</th>
<th>$F(x_k) - F^*$</th>
<th>$|x_k|_0$</th>
<th>time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0010</td>
<td>$&lt; 10^{16}$</td>
<td>857</td>
<td>0.01</td>
</tr>
<tr>
<td>12.72</td>
<td>$&lt; 10^{11}$</td>
<td>999,246</td>
<td>54.48</td>
</tr>
<tr>
<td>15.23</td>
<td>$&lt; 10^{10}$</td>
<td>997,944</td>
<td>65.19</td>
</tr>
<tr>
<td>18.61</td>
<td>$&lt; 10^9$</td>
<td>990,923</td>
<td>79.69</td>
</tr>
<tr>
<td>20.61</td>
<td>$&lt; 10^8$</td>
<td>978,761</td>
<td>88.25</td>
</tr>
<tr>
<td>23.38</td>
<td>$&lt; 10^7$</td>
<td>926,167</td>
<td>100.08</td>
</tr>
<tr>
<td>25.91</td>
<td>$&lt; 10^6$</td>
<td>763,314</td>
<td>110.94</td>
</tr>
<tr>
<td>28.22</td>
<td>$&lt; 10^5$</td>
<td>366,835</td>
<td>120.84</td>
</tr>
<tr>
<td>30.66</td>
<td>$&lt; 10^4$</td>
<td>57,991</td>
<td>131.25</td>
</tr>
<tr>
<td>32.83</td>
<td>$&lt; 10^3$</td>
<td>9,701</td>
<td>140.55</td>
</tr>
<tr>
<td>35.05</td>
<td>$&lt; 10^2$</td>
<td>2,538</td>
<td>150.02</td>
</tr>
<tr>
<td>37.01</td>
<td>$&lt; 10^1$</td>
<td>1,722</td>
<td>158.39</td>
</tr>
<tr>
<td>38.26</td>
<td>$&lt; 10^0$</td>
<td>1,633</td>
<td>163.75</td>
</tr>
<tr>
<td>40.98</td>
<td>$&lt; 10^{-1}$</td>
<td>1,604</td>
<td>175.38</td>
</tr>
<tr>
<td>42.71</td>
<td>$&lt; 10^{-3}$</td>
<td>1,600</td>
<td>182.75</td>
</tr>
<tr>
<td>42.71</td>
<td>$&lt; 10^{-4}$</td>
<td>1,600</td>
<td>182.77</td>
</tr>
<tr>
<td>43.28</td>
<td>$&lt; 10^{-5}$</td>
<td>1,600</td>
<td>185.21</td>
</tr>
<tr>
<td>44.86</td>
<td>$&lt; 10^{-6}$</td>
<td>1,600</td>
<td>191.94</td>
</tr>
</tbody>
</table>
**Instance 2 run on CPU: \( A \in \mathbb{R}^{10^9 \times 10^8} \), \( \| A \|_0 = 2 \times 10^9 \)

<table>
<thead>
<tr>
<th>( k/N )</th>
<th>( F(x_k) - F^* )</th>
<th>( | x_k |_0 )</th>
<th>time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>(&lt; 10^{18})</td>
<td>18,486</td>
<td>1.32</td>
</tr>
<tr>
<td>9.35</td>
<td>(&lt; 10^{14})</td>
<td>99,837,255</td>
<td>1294.72</td>
</tr>
<tr>
<td>11.97</td>
<td>(&lt; 10^{13})</td>
<td>99,567,891</td>
<td>1657.32</td>
</tr>
<tr>
<td>14.78</td>
<td>(&lt; 10^{12})</td>
<td>98,630,735</td>
<td>2045.53</td>
</tr>
<tr>
<td>17.12</td>
<td>(&lt; 10^{11})</td>
<td>96,305,090</td>
<td>2370.07</td>
</tr>
<tr>
<td>20.09</td>
<td>(&lt; 10^{10})</td>
<td>86,242,708</td>
<td>2781.11</td>
</tr>
<tr>
<td>22.60</td>
<td>(&lt; 10^{9})</td>
<td>58,157,883</td>
<td>3128.49</td>
</tr>
<tr>
<td>24.97</td>
<td>(&lt; 10^{8})</td>
<td>19,926,459</td>
<td>3455.80</td>
</tr>
<tr>
<td>28.62</td>
<td>(&lt; 10^{7})</td>
<td>747,104</td>
<td>3960.96</td>
</tr>
<tr>
<td>31.47</td>
<td>(&lt; 10^{6})</td>
<td>266,180</td>
<td>4325.60</td>
</tr>
<tr>
<td>34.47</td>
<td>(&lt; 10^{5})</td>
<td>175,981</td>
<td>4693.44</td>
</tr>
<tr>
<td>36.84</td>
<td>(&lt; 10^{4})</td>
<td>163,297</td>
<td>5004.24</td>
</tr>
<tr>
<td>39.39</td>
<td>(&lt; 10^{3})</td>
<td>160,516</td>
<td>5347.71</td>
</tr>
<tr>
<td>41.08</td>
<td>(&lt; 10^{2})</td>
<td>160,138</td>
<td>5577.22</td>
</tr>
<tr>
<td>43.88</td>
<td>(&lt; 10^{1})</td>
<td>160,011</td>
<td>5941.72</td>
</tr>
<tr>
<td>45.94</td>
<td>(&lt; 10^{0})</td>
<td>160,002</td>
<td>6218.82</td>
</tr>
<tr>
<td>46.19</td>
<td>(&lt; 10^{-1})</td>
<td>160,001</td>
<td>6252.20</td>
</tr>
<tr>
<td>46.25</td>
<td>(&lt; 10^{-2})</td>
<td>160,000</td>
<td>6260.20</td>
</tr>
<tr>
<td>46.89</td>
<td>(&lt; 10^{-3})</td>
<td>160,000</td>
<td>6344.31</td>
</tr>
<tr>
<td>46.91</td>
<td>(&lt; 10^{-4})</td>
<td>160,000</td>
<td>6346.99</td>
</tr>
<tr>
<td>46.93</td>
<td>(&lt; 10^{-5})</td>
<td>160,000</td>
<td>6349.69</td>
</tr>
</tbody>
</table>
GPU (Graphical Processing Unit): performs a single operation in parallel on multiple data (vector addition).
GPU Architecture

Tesla C2050

- 14 Multiprocessors
- Each Multiprocessor has 32 cores (each 1.15GHz)
- Each core in one multiprocessor has to do the same operation on different data = SIMD (Single Instruction Multiple Data)
Huge and Sparse Data = Small Chance of Collision
Race Conditions and Atomic Operation

**Race Conditions**

- Shared var = 10
  - Thread A reads
    - Add 10
    - Shared var = 20
      - Shared var = 20
      - Shared var = 15
      - Final value
  - Thread B reads
    - Add 5
    - Shared var = 15

**CUDA Atomic Operation**

- Shared var = 10
  - Thread A reads
    - Add 10
    - Shared var = 20
  - Thread B reads
    - Add 5
    - Shared var = 25
    - Final value

GPU
Serial Algorithm

1. Choose (block) coordinate i
2. Compute update T
3. Update x

Parallel Algorithm

1. Choose (block) coordinate i
2. Compute update T
3. Update x
CPU (C) vs GPU (CUDA): step-by-step comparison

\[ n = 10^8 \]
Acceleration by GPU

![Graph showing the comparison between GPU and CPU performance over time.](image-url)
Naive Implementation

Implementation

- each thread choose its coordinate independently,
- matrix $A$ is naturally stored in CSC sparse format.
Naive Implementation

Data access

- is NOT coalesced (is NOT sequential at all)
- is misaligned (decrease of performance)

### Misaligned and sequential

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td>...</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compute capability:</th>
<th>1.0 and 1.1</th>
<th>1.2 and 1.3</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory transactions:</td>
<td>Uncached</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 x 32B at 128</td>
<td>7 x 32B at 128</td>
<td>1 x 128B at 128</td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 160</td>
<td>8 x 32B at 160</td>
<td>1 x 64B at 192</td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 192</td>
<td>8 x 32B at 192</td>
<td>1 x 32B at 256</td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 224</td>
<td>8 x 32B at 224</td>
<td>1 x 128B at 256</td>
</tr>
<tr>
<td></td>
<td>1 x 32B at 256</td>
<td>1 x 32B at 256</td>
<td>1 x 128B at 256</td>
</tr>
</tbody>
</table>
__global__ void offsetCopy(float* odata, float* idata, int offset) {
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[xid];
}
Better implementation

Memory access improvement

- all threads in ONE WRAP generate the same random number which points to begin of memory bank (from where data will be read)
- each thread then compute its coordinate \(i\) (one wrap is working on ALIGNED and SEQUENTIAL coordinates)
Pipelining

C++ code

```c++
int offset = ....;
for (int i = -offset; i < N; i++) {
}
```

**Left:** Offset is a constant. **Right:** Offset is a variable, therefore compiler cannot optimize as well as previously.
Facing the Multicore-Challenge

II

(a few insights from a conference)
SIMD and Auto-Vectorization

**Figure 1.8:** Example for SIMD: Single precision FP addition of two SIMD registers \((x,y)\), each having a length of 128 bits. Four SP flops are executed in a single instruction.

**CPU**

2.2 GHz, 4 flops/cycle, 12 cores = **105.6** GFlops (AMD Magny-Cours)

**GPU**

**1030.4** GFlops (Tesla C2050/C2070 GPU Computing Processor)

...the memory subsystem (or at least the cache) must be able to sustain sufficient bandwidth to keep all units busy...
-guide=n
analyzes code and gives some recommendations
-vec-report⟨n⟩
report of loop vectorization, shows which loops were vectorized and which not
#pragma ivdep
vectorization, do not care on data dependences
-ax
auto-vectorization

C++ code

```cpp
int* C[N];
....
for (int j = 0; j < C[7]; j++) {
    //C[7]=2034
    A[j]=2*B[j];
}
```
-guide=n
analyzes code and gives some recommendations
-vec-report\langle n\rangle
report of loop vectorization, shows which loops where vectorized and which not
#pragma ivdep
vectorization, do not care on data dependences
#pragma ivdep
auto-vectorization

C++ code

```cpp
int* C[N];

#pragma ivdep
for (int j = 0; j < C[7]; j++) {
    A[j] = 2 * B[j];
    //C[7]=2034
}
```

```c
#pragma css task input(X) inout(Y)
void work(float X[N][N], float Y[N][N])

void main(int argc, char* argv[]){
    ...
    #pragma css start
    for (i=0; i<M; i++){
        work(A, B); ①
        work(A, C); ②
        work(B, D); ③
        work(C, A); ④
    }
    #pragma css finish
}
```

- **Dependences**

  - **RaW** (Read after Write) → Dependence between tasks
  - **WaR** (Write after Read) → Dependence that can be solved by renaming
**RaW** (Read after Write) → Dependence between tasks

**War** (Write after Read) → Dependence that can be solved by renaming

1 must run after 3. Because we cannot modify the value of A in 4 before 1 reads it

1 has a copy of A with its correct value. There is no dependence between 1 and 4.
#pragma css task input(X) inout(Y)
void work(float X[N][N], float Y[N][N])

#pragma css task target device(gpu)
   input(X) inout(Y)
void work_gpu(float X[N][N], float Y[N][N])

void main(int argc, char* argv[]){
  ...
  #pragma css start
  for (i=0; i<M; i++){
    work(A, B);  ①
    work_gpu(A, C);  ②
    work(B, D);  ③
    work_gpu(C, A);  ④
  }
  #pragma css finish
}
```c
#pragma css task input(X) inout(Y)
void work(float X[N][N], float Y[N][N])
{
    ...
    #pragma css start
    for (i=0; i<M; i++){
        work(A, B); 1
        work(A, C); 2
        work(B, D); 3
        work(C, A); 4
    }
    #pragma css finish
}
```
Write parallel programs using a simple model: With only three keywords to learn, C and C++ developers move quickly into the parallel programming domain.

Utilize data parallelism by simple array notations that include elemental function capabilities.

Leverage existing serial tools: The serial semantics of Intel Cilk Plus allows you to debug in a familiar serial debugger.

Scale for the future: The runtime system operates smoothly on systems with hundreds of cores.
```c
__declspec(vector) float saxpy_elemental(float a, float x, float y)
{
    return (a * x + y);
}

// Here is how to invoke such vectorized elemental functions using array notation:
z[:] = saxpy_elemental(a, b[:], c[:]);

a[:] = sin(b[:]); // vector math library function
a[:] = pow(b[:], c); // b[:]**c via a vector math library function
```
cilk int fib (int n) {
    if (n < 2) return n;
    else {
        int x, y;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync;
        return (x+y);
    }
}
Thank you for your attention
References

- Georg Hager and Gerhard Wellein: Introduction to High Performance Computing for Scientists and Engineers
- Jesus Labarta, Rosa Badia, Eduard Ayguade, Marta Garcia, Vladimir Marjanovic: Hybrid Parallel Programming with MPI+SMPSs