Paralelisasi Metode Jacobi untuk Komputasi SVD dalam GPU dan Modifikasinya

Adi Sujiwo

Daftar Isi

Pendahuluan

Pengantar

- Komputasi SVD untuk matriks besar berjalan lambat
- Metode klasik untuk SVD: Golub-Kahan-Reisch
- Diperlukan cara-cara inovatif untuk mempercepatnya
- Paralelisasi SVD dalam GPU
- Metode Jacobi terbukti paling mudah diparalelisasi

Executive Summary

- Tujuan Penelitian: **menyelidiki kemungkinan akselerasi SVD, khususnya metode Jacobi, dengan menggunakan GPU**
- Penulis membangun:
  - Rutin SVD dengan metode One-Side Block Jacobi (OSBJA) dalam GPU
  - Urutan ortogonalisasi baru: berdasarkan anti-diagonal
  - Rutin dekomposisi QR dengan metode Rotasi Givens dalam GPU
  - Kombinasi Jacobi dengan QR dalam GPU
- Hasil:
  - Rutin SVD dengan metode Jacobi dan prekondisi QR mampu mengungguli Octave pada matriks berukuran besar

Kontribusi

- Paralelisasi Jacobian SVD dalam GPU
- Metode pengurutan ortogonalisasi per-anti-diagonal matriks pasangan baris
- Implementasi QR dengan rotasi Givens dalam GPU

Mengapa GPU ?

- Murah-Meriah
- Tersedia luas (semua PC pasti memiliki GPU)
- Cocok untuk aplikasi data-parallel
- Embedded GPU sudah tersedia

Penelitian Sebelumnya

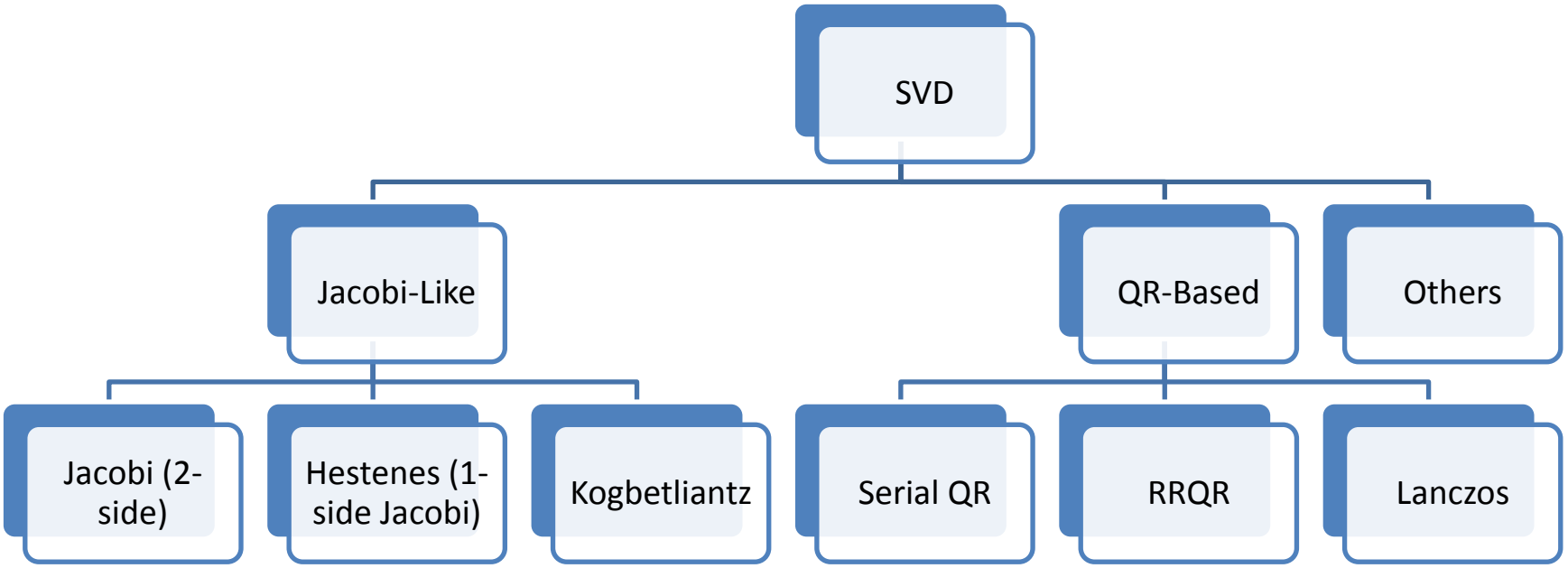
- Franklin Luk (1989) : pembuktian konvergensi metode Jacobi
- Strumpen et. al. (2003) : implementasi algoritma Jacobi pada prosesor stream
- Hari (2005) : modifikasi algoritma Jacobi dengan prekondisi QR
- Lahabar et. al. (2008) mengimplementasikan SVD dengan metode Golub-Kahan pada GPU
- Shane Ryoo et. al (2008) : prinsip-prinsip optimisasi dan evaluasi aplikasi GPU

Asumsi-Asumsi Dasar

- Matriks sumber adalah $M \in \mathbb{R}^{m \times n}$
- $m \geq n$
- Format penyimpanan matriks dalam memori adalah row-major
- Presisi tunggal
- Indeks array dimulai dari 0

Tinjauan Pustaka

Klasifikasi Algoritma SVD



Two-Side Jacobi

- Dasar: diagonalisasi matriks simetriks melalui perkalian dengan matriks rotasi dari kiri dan kanan

$$A^{k+1} = J(i, j, \theta) A^k J(i, j, \theta)^T$$

- Update pada baris dan kolom ke-i dan j → dapat diparalelkan
- Kerugian: harus bekerja dengan baris dan kolom secara bersamaan

Kogbetliantz Method

- Dasar: diagonalisasi matriks segitiga $A \in \mathbb{R}^{n \times n}$ dengan perkalian dua matriks rotasi berbeda dari kiri dan kanan

$$A^{k+1} = J_1(i, j, \theta) A^k J_2(i, j, \phi)^T$$

- Seperti two-side Jacobi, harus bekerja pada baris dan kolom bersamaan
- Hari (2007) menyarankan transformasi matriks menjadi bentuk “Butterfly” supaya dapat bekerja dalam baris atau kolom saja.

Metode SVD: One-Side Block Jacobi

- Dasar: Rotasi setiap pasang vektor baris sampai saling ortogonal
- Memerlukan beberapa kali "*sweep*" sampai tercapai ortogonalitas ("konvergen").
- Menggunakan rotasi Jacobi untuk membuat dua vektor ortogonal
- Mengalikan matriks dengan matriks rotasi dari sisi kiri ("one-side")
- Dilakukan per pasangan blok-blok baris berukuran tetap ("block")

Penurunan OSBJA

Iterasi Rotasi

Matriks rotasi→ortogonal

$$A^{(t)} = \Phi^{(t)}\Phi^{(t-1)} \dots \Phi^{(2)}\Phi^{(1)}A$$

Sampai didapat baris-baris $A^{(t)}$ saling ortogonal

Rotasi dapat diakumulasi sebagai

$$U^T = \Phi^{(t)}\Phi^{(t-1)} \dots \Phi^{(2)}\Phi^{(1)}I$$

Perkalian rotasi→ortogonal

Sehingga

$$A^{(t)} = U^T A$$
$$= \Sigma V^T$$

Nilai-nilai singular dari A:

$$s_{i,i} = \left\| A^{(t)}[i] \right\|$$

$$V^T[i] = \frac{A^{(t)}[i]}{s_{i,i}}$$

Output:

Matriks $U^T \in \mathbb{R}^{m \times m}$

Vektor $S \in \mathbb{R}^n$

Matriks $V^T \in \mathbb{R}^{m \times m}$

Formula Rotasi

Matriks Rotasi:

$$\Phi(i,j,\theta) = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & c & & s & & \\ & & & \ddots & & & \\ & & -s & & c & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix}$$

Di mana $c = \cos\theta$ dan $s = \sin\theta$

Menghitung c dan s:

$$w = \frac{\|A[j]\| - \|A[i]\|}{2(A^T[i] \cdot A[j])}$$

$$t = \frac{\text{sign}(w)}{|w| + \sqrt{1 + w^2}}$$

$$c = \frac{1}{\sqrt{1 + t^2}}$$

$$s = tc$$

Nilai baru baris-baris A setelah rotasi adalah

$$A[i]_{\text{new}} = cA[i] - sA[j]$$

$$A[j]_{\text{new}} = sA[i] + sA[j]$$

Kriteria Konvergensi Jacobi

Strumpen et. al (2003)

$$\delta = \epsilon \sum_{i}^{m} \|A[i]\|$$

*Dipilih untuk percobaan ini

HPEC Challenge (2006)

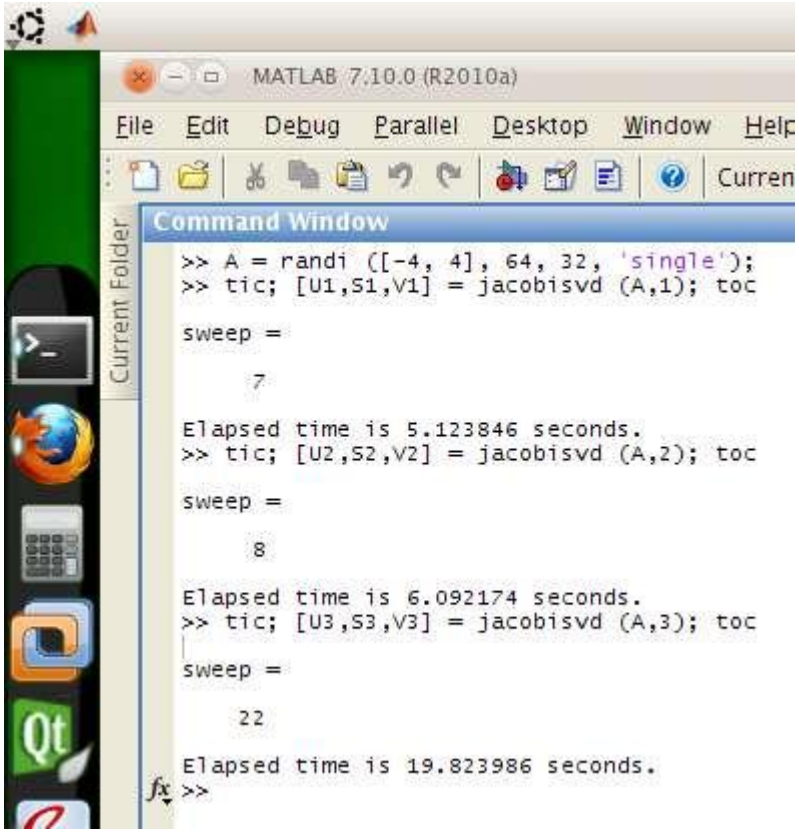
$$\delta = 10\epsilon \cdot \max(m, n)$$

Urutan Ortogonalisasi

- Urutan ortogonalisasi baris mempengaruhi jumlah sweep →kecepatan konvergensi
- Tiga jenis urutan:
  - Serial cyclic-by-row/column
  - Paralel (Brent & Luk)
  - Odd-Even
- Belum ada pembuktian apakah urutan tertentu (selain serial) dijamin konvergen

Brent-Luk Parallel Ordering

- Dapat dikerjakan oleh $\lceil m/2 \rceil$ jumlah prosesor secara serentak
- Hasil eksperimen: perlu terlalu banyak sweep jika dibandingkan urutan serial (untuk one-side)



Langkah 1

1	3	5	7
2	4	6	8

Langkah 2

1	2	3	5
4	6	8	7

Langkah 3

1	4	2	3
6	8	7	5

Langkah 4

1	6	4	2
8	7	5	3

Langkah 5

1	8	6	4
7	5	3	2

Langkah 6

1	7	8	6
5	3	2	4

Langkah 7

1	5	7	8
3	2	4	6

Kunggulan dan Kelemahan Metode Jacobi

- Keunggulan:
  - Sederhana, elegan
  - Akurat, mampu menghitung nilai-nilai singular yang kecil
  - Potensial diparalelisasi
- Kelemahan:
  - It is surely **SLOW**

Prinsip-Prinsip Perbaikan Metode Jacobi (Hari 2005)

- Pengurangan jumlah langkah dan sweep
- Pengurangan jumlah flop dan waktu CPU dalam satu langkah
- Perbaikan detil implementasi: eksploitasi cache, improvisasi dot products, dll.

Modifikasi SVD dengan QR

- Prekondisi matriks sumber untuk mengurangi jumlah baris yang harus di-ortogonalisasi
- Berguna jika $m \gg n$
- Algoritma:
  1. Dekomposisi QR: $A = Q_1 R$
  2. Dekomposisi LQ: $R = L Q_2^T$
  3. Hitung SVD: $L = U_1 \Sigma V_1^T$
  4. SVD dari A: $A = (Q_1 U_1) \Sigma (Q_2 V_1)^T$

Potong R berukuran $n \times n$

Perbesar U_1 menjadi $m \times m$

Metode QR: Rotasi Givens

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

X	X	X
X	X	X
X	X	X
X	X	X

Input

skip

Metode QR: Rotasi Givens

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

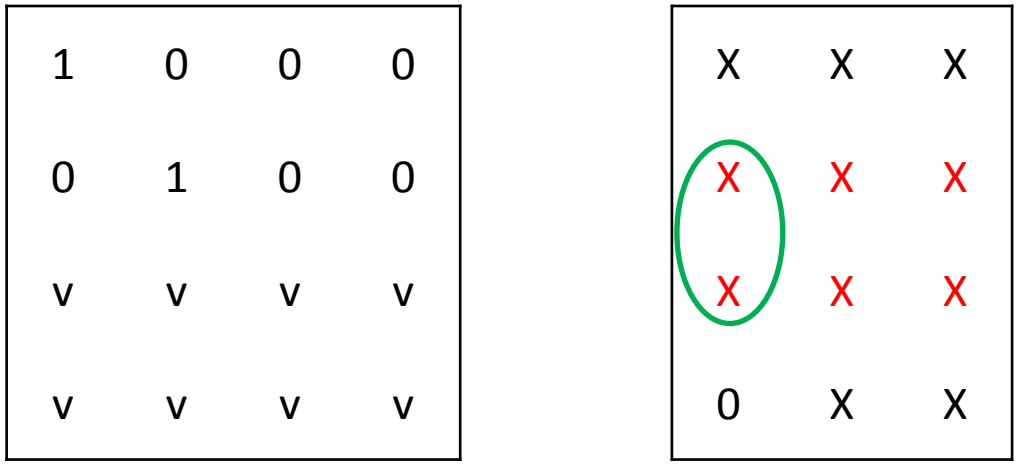
X	X	X
X	X	X
X	X	X
X	X	X

Metode QR: Rotasi Givens

1	0	0	0
0	1	0	0
v	v	v	v
v	v	v	v

X	X	X
X	X	X
X	X	X
0	X	X

Metode QR: Rotasi Givens

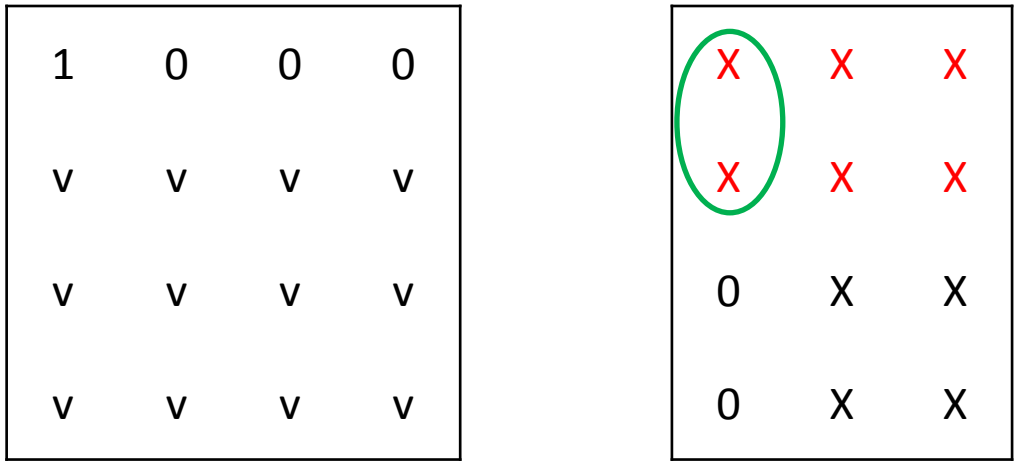


Metode QR: Rotasi Givens

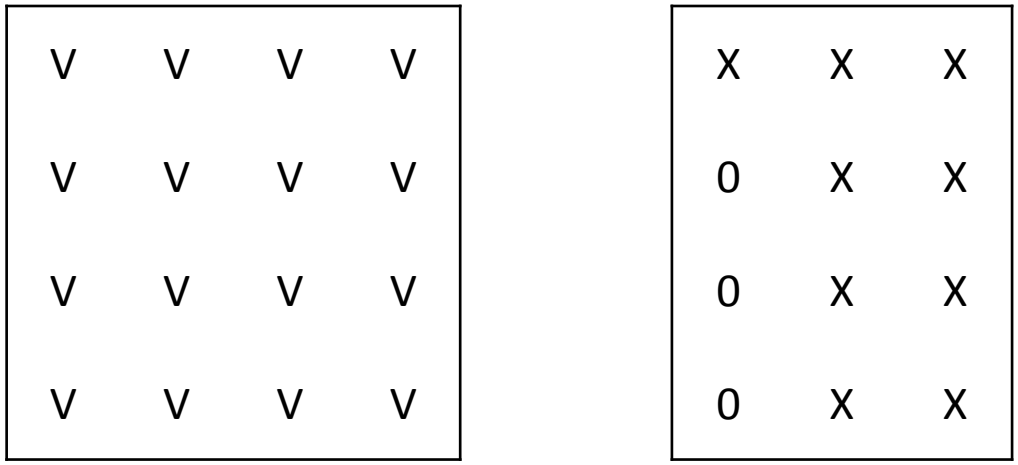
1	0	0	0
v	v	v	v
v	v	v	v
v	v	v	v

X	X	X
X	X	X
0	X	X
0	X	X

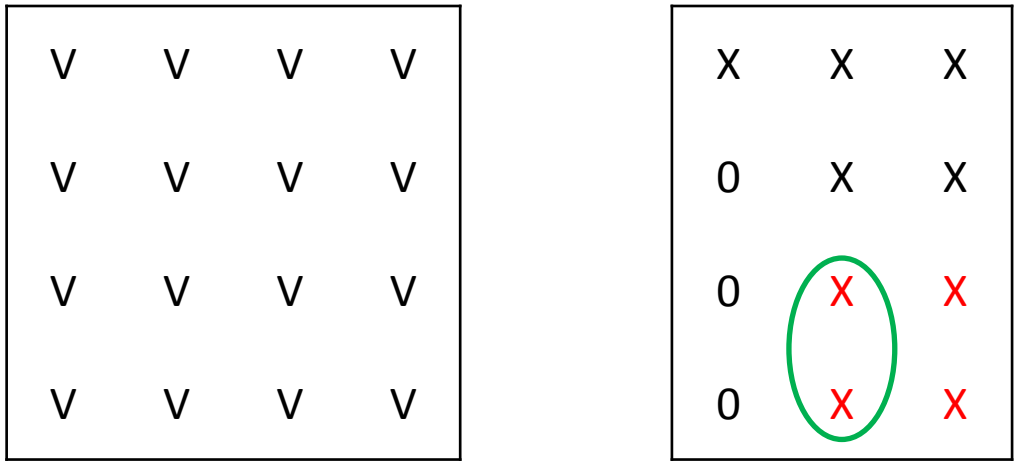
Metode QR: Rotasi Givens



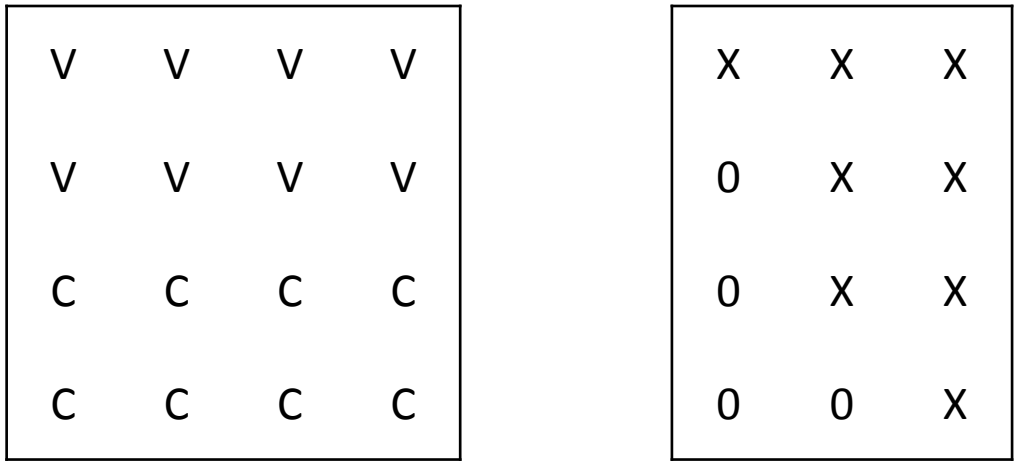
Metode QR: Rotasi Givens



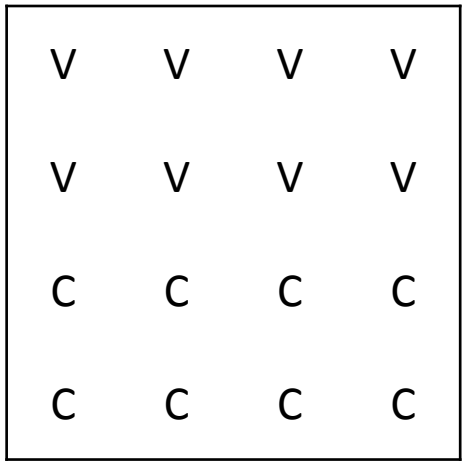
Metode QR: Rotasi Givens

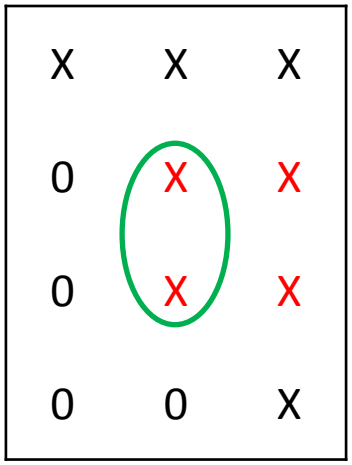


Metode QR: Rotasi Givens



Metode QR: Rotasi Givens



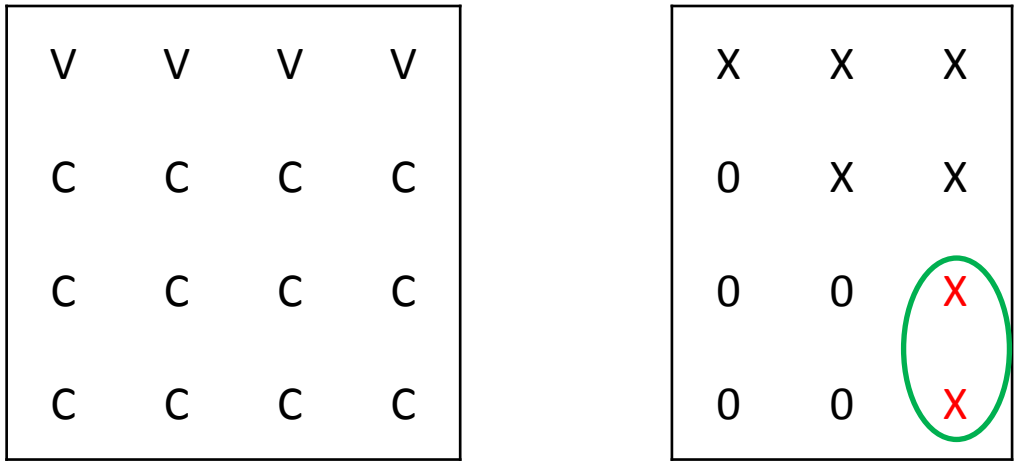


Metode QR: Rotasi Givens

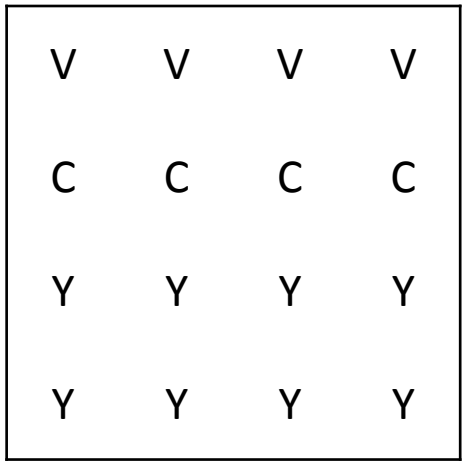
v	v	v	v
c	c	c	c
c	c	c	c
c	c	c	c

x	x	x
0	x	x
0	0	x
0	0	x

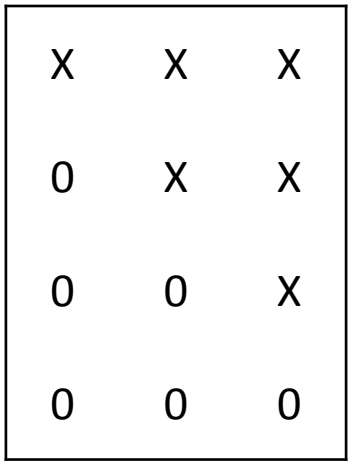
Metode QR: Rotasi Givens



Metode QR: Rotasi Givens

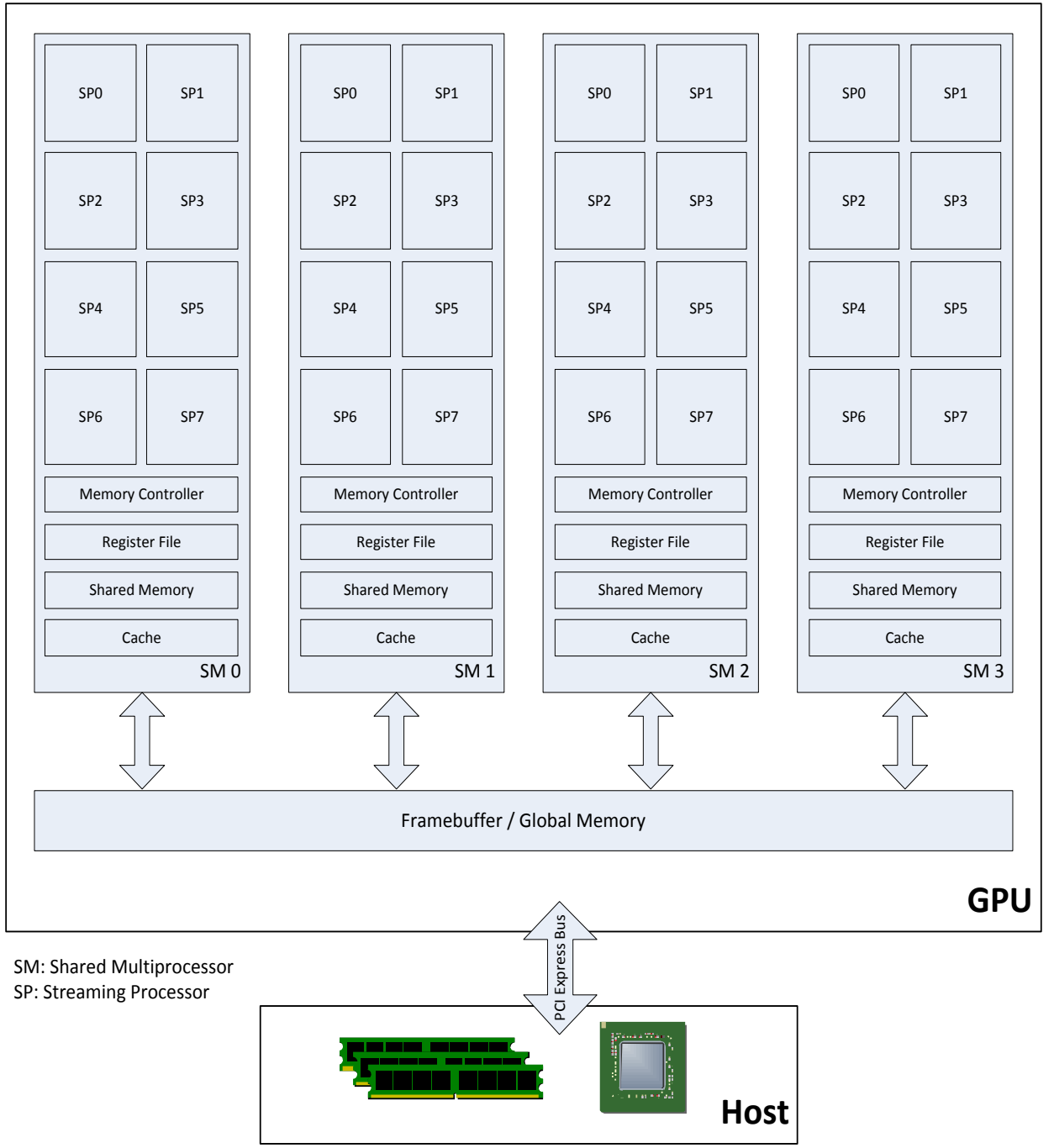


Q’



R

Struktur GPU



SM: Shared Multiprocessor
SP: Streaming Processor

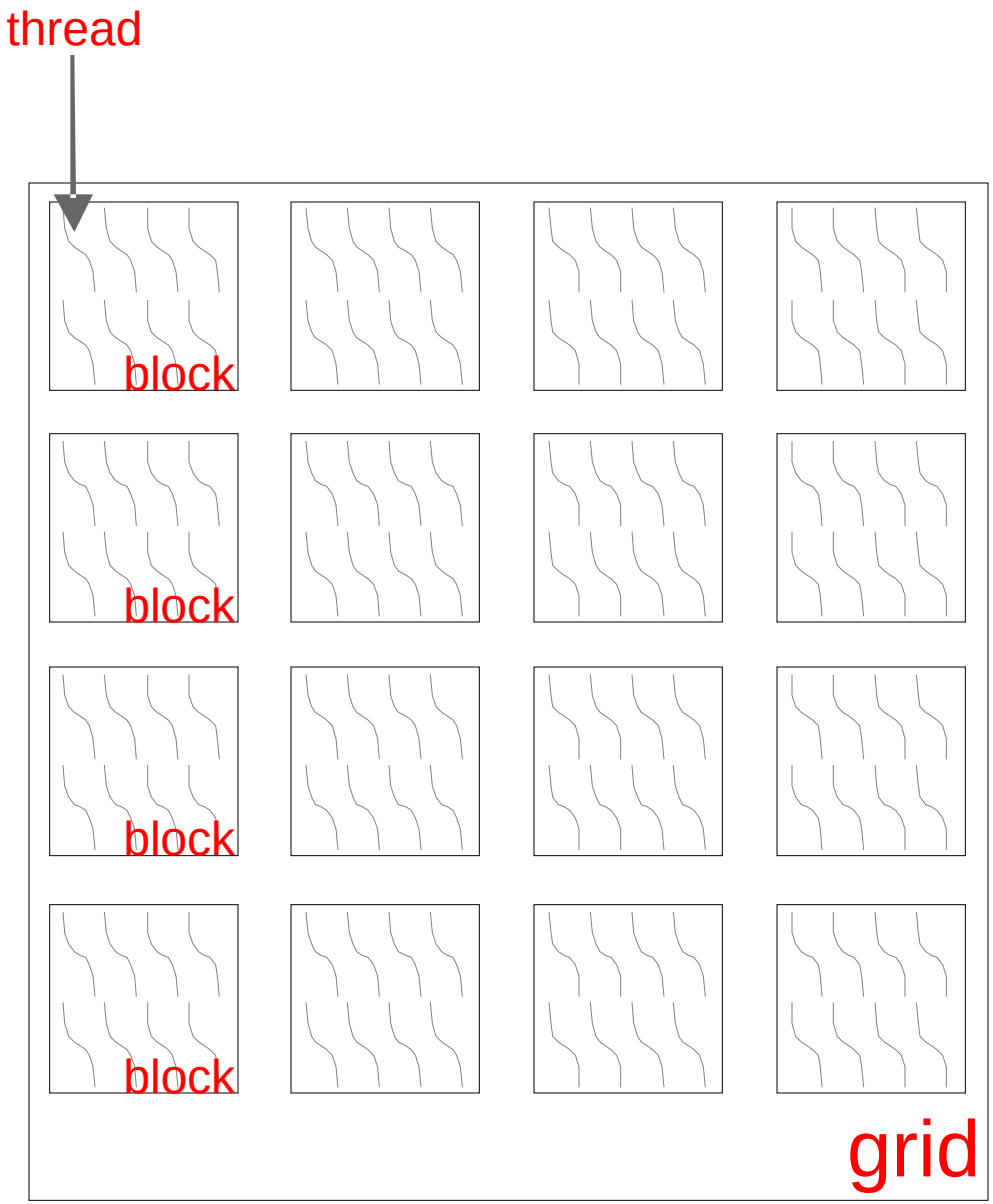
CUDA: Perangkat Pemrograman GPU

- Kode konvensional yang berjalan dalam CPU melakukan panggilan kernel yang berjalan pada GPU
- Panggilan kernel disertai konfigurasi eksekusi: ukuran *grid*, *block*
- Kode dalam kernel berjalan *per-thread*
- *Thread* →*block*→*grid*
- Sinkronisasi antar *thread* dalam *block* (tidak bisa antar *block* atau global)
- Kerjasama antar thread dalam block: sinkronisasi, shared memory, *warp*.

Hirarki:

Thread→Block→Grid

- Satu block berjalan pada satu SM
- Beberapa block dapat berjalan bersama pada satu SM, dibatasi oleh konsumsi register dan shared memory→tetap tidak ada sinkronisasi antar block→bayangkan OS multitasking pada uniprosesor
- **Grid**: kumpulan semua block yang berada dalam GPU
- **Warp**: kelompok 32 thread yang berjalan serentak pada satu block
- *Divergent warp*: percabangan yang membuat anggota warp mengambil jalur kode berbeda
- Divergent warp memiliki ongkos yang mahal: setiap cabang harus diselesaikan secara serial
- Percabangan antar warp tidak mendapat penalty.
- 8 SP untuk menjalankan 32 thread→satu instruksi sederhana perlu minimal 4 *clock cycle*



Pendekatan Pemrograman GPU

- *Data-Parallelism*
- Kode CPU sebagai sinkronisasi, GPU sebagai pemrosesan
- Implisit: pandang blok sebagai *vector processor*, gunakan *shared memory* sebagai penyimpan vektor sesering mungkin
- Koordinasikan akses memori dengan sesama thread dalam block
- Minimalisir percabangan

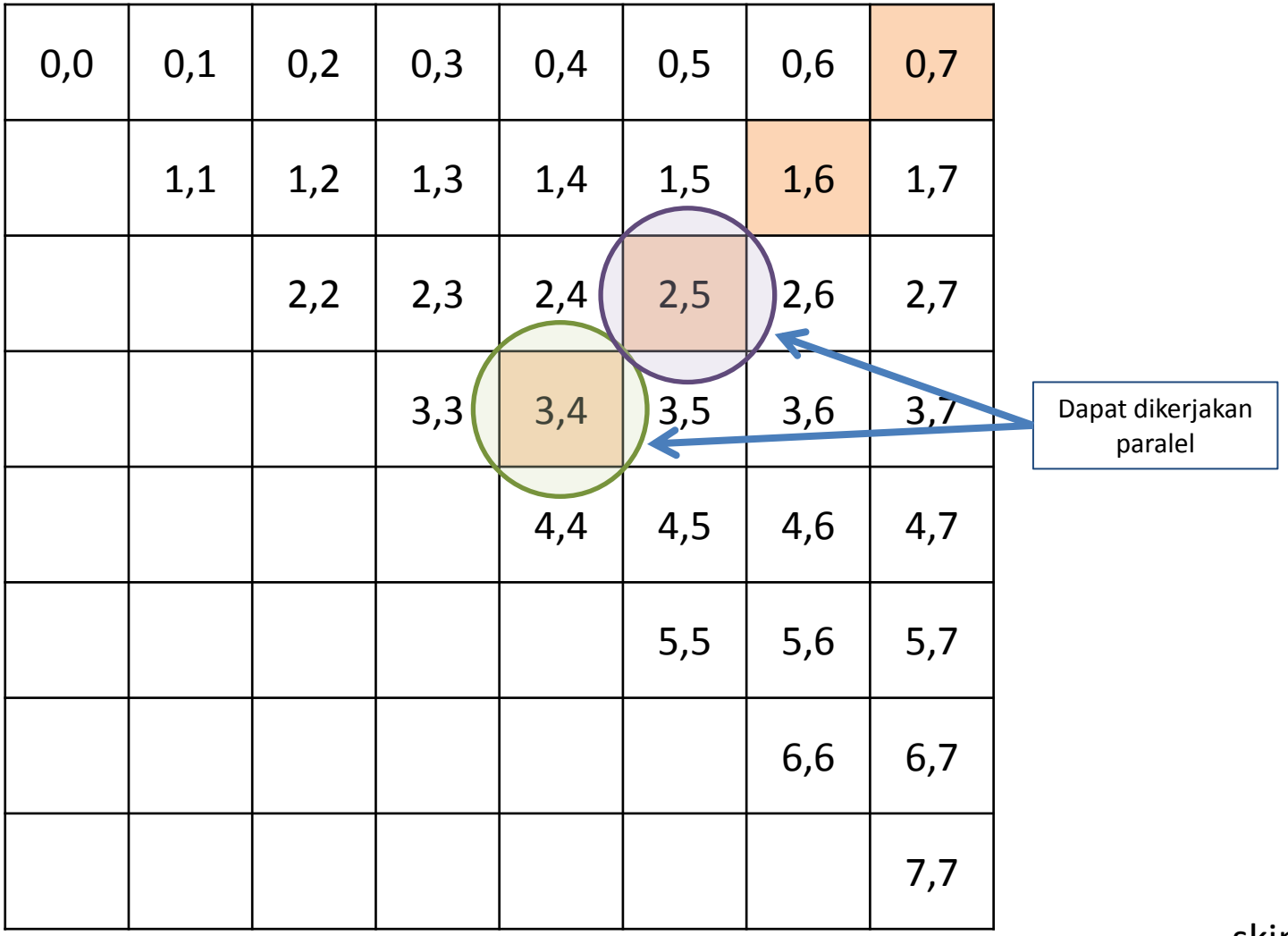
Rancangan Algoritma

Paralelisasi OSBJA dalam GPU

- Perlu $m(m-1)/2$ proses ortogonalisasi per sweep
- Urutan kerja: setiap kali proses tidak boleh ada satu baris yang diproses bersamaan
- Bagi baris-baris menjadi k kelompok berukuran sama, buat pasangan setiap kelompok dan didapat sebuah matriks pasangan kelompok baris
- Tidak semua urutan pengerjaan dijamin konvergen
- Solusi: Urutan ortogonalisasi menurut anti-diagonal
- Setiap pasangan blok dalam satu anti-diagonal dapat dikerjakan paralel

Urutan Ortogonalisasi Menurut Anti-Diagonal

Langkah 0



skip

Urutan Pengerjaan Menurut Anti-Diagonal

Langkah 1A

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
	1,1	1,2	1,3	1,4	1,5	1,6	1,7
		2,2	2,3	2,4	2,5	2,6	2,7
			3,3	3,4	3,5	3,6	3,7
				4,4	4,5	4,6	4,7
					5,5	5,6	5,7
						6,6	6,7
							7,7

Urutan Pengerjaan Menurut Anti-Diagonal

Langkah 1A

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
	1,1	1,2	1,3	1,4	1,5	1,6	1,7
		2,2	2,3	2,4	2,5	2,6	2,7
			3,3	3,4	3,5	3,6	3,7
				4,4	4,5	4,6	4,7
					5,5	5,6	5,7
						6,6	6,7
							7,7

Urutan Pengerjaan Menurut Anti-Diagonal

Langkah 1B

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
	1,1	1,2	1,3	1,4	1,5	1,6	1,7
		2,2	2,3	2,4	2,5	2,6	2,7
			3,3	3,4	3,5	3,6	3,7
				4,4	4,5	4,6	4,7
					5,5	5,6	5,7
						6,6	6,7
							7,7

Urutan Pengerjaan Menurut Anti-Diagonal

Langkah 1B

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
	1,1	1,2	1,3	1,4	1,5	1,6	1,7
		2,2	2,3	2,4	2,5	2,6	2,7
			3,3	3,4	3,5	3,6	3,7
				4,4	4,5	4,6	4,7
					5,5	5,6	5,7
						6,6	6,7
							7,7

Urutan Pengerjaan Menurut Anti-Diagonal

Langkah 2A

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
	1,1	1,2	1,3	1,4	1,5	1,6	1,7
		2,2	2,3	2,4	2,5	2,6	2,7
			3,3	3,4	3,5	3,6	3,7
				4,4	4,5	4,6	4,7
					5,5	5,6	5,7
						6,6	6,7
							7,7

Urutan Pengerjaan Menurut Anti-Diagonal

Langkah 2B

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
	1,1	1,2	1,3	1,4	1,5	1,6	1,7
		2,2	2,3	2,4	2,5	2,6	2,7
			3,3	3,4	3,5	3,6	3,7
				4,4	4,5	4,6	4,7
					5,5	5,6	5,7
						6,6	6,7
							7,7

Urutan Pengerjaan Menurut Anti-Diagonal

Langkah 3

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
	1,1	1,2	1,3	1,4	1,5	1,6	1,7
		2,2	2,3	2,4	2,5	2,6	2,7
			3,3	3,4	3,5	3,6	3,7
				4,4	4,5	4,6	4,7
					5,5	5,6	5,7
						6,6	6,7
							7,7

Urutan Pengerjaan Menurut Anti-Diagonal

Langkah 3

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
	1,1	1,2	1,3	1,4	1,5	1,6	1,7
		2,2	2,3	2,4	2,5	2,6	2,7
			3,3	3,4	3,5	3,6	3,7
				4,4	4,5	4,6	4,7
					5,5	5,6	5,7
						6,6	6,7
							7,7

Urutan Pengerjaan Menurut Anti-Diagonal

Langkah 3

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
	1,1	1,2	1,3	1,4	1,5	1,6	1,7
		2,2	2,3	2,4	2,5	2,6	2,7
			3,3	3,4	3,5	3,6	3,7
				4,4	4,5	4,6	4,7
					5,5	5,6	5,7
						6,6	6,7
							7,7

Urutan Pengerjaan Menurut Anti-Diagonal

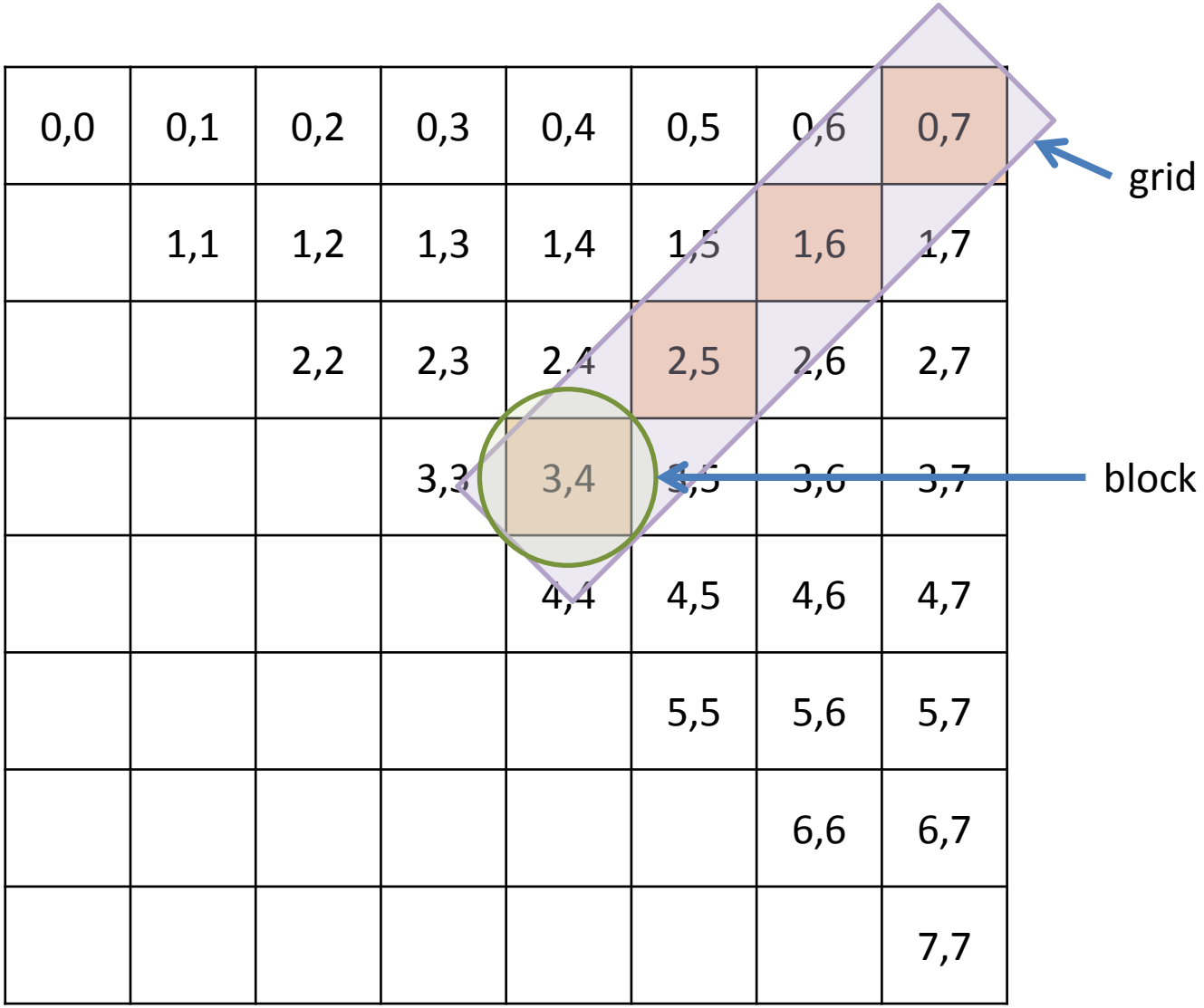
Langkah 3

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
	1,1	1,2	1,3	1,4	1,5	1,6	1,7
		2,2	2,3	2,4	2,5	2,6	2,7
			3,3	3,4	3,5	3,6	3,7
				4,4	4,5	4,6	4,7
					5,5	5,6	5,7
						6,6	6,7
							7,7

Implementasi Jacobi dalam CUDA

- Kode sinkronisasi antar anti-diagonal dalam CPU
- Kode kernel untuk ortogonalisasi sepasang blok baris→dipetakan ke satu block GPU
- Satu block GPU berisi 64 thread, satu blok baris berisi dua baris

Pembagian Kerja Thread, Block & Grid



Kernel Jacobi

Tahap 1: Menghitung *dot products*

- $r_1 = \mathbf{l}_1 \cdot \mathbf{l}_1$
- $r_2 = \mathbf{l}_2 \cdot \mathbf{l}_2$
- $r_3 = \mathbf{l}_1 \cdot \mathbf{l}_2$

"Loop Unroll"
Syarat jumlah baris harus kelipatan $b = 64$

Akumulasi dot products

```
for (i=0:⌈n/b⌉ − 1) {
    row1 [tid] = A [l1*n + i*b + tid];
    row2 [tid] = A [l2*n + i*b + tid];
    row3 [tid] = row1 [tid] * row2 [tid];
    row2 [tid] = row2 [tid] * row2 [tid];
    row1 [tid] = row1 [tid] * row1 [tid];
    __syncthreads ();

    if (tid < 32) {
        row1 [tid] += row1 [tid + 32];
        row1 [tid] += row1 [tid + 16];
        row1 [tid] += row1 [tid +  8];
        row1 [tid] += row1 [tid +  4];
        row1 [tid] += row1 [tid +  2];
        row1 [tid] += row1 [tid +  1];
        row3 [tid] += row3 [tid + 32];
        row3 [tid] += row3 [tid + 16];
        row3 [tid] += row3 [tid +  8];
        row3 [tid] += row3 [tid +  4];
        row3 [tid] += row3 [tid +  2];
        row3 [tid] += row3 [tid +  1];
    }

    else {
        row2 [tid] += row2 [tid + 32];
        row2 [tid] += row2 [tid + 16];
        row2 [tid] += row2 [tid +  8];
        row2 [tid] += row2 [tid +  4];
        row2 [tid] += row2 [tid +  2];
        row2 [tid] += row2 [tid +  1];
    }

    if (tid==0) {
        row1 [b] += row1 [0];
        row2 [b] += row2 [0];
        row3 [b] += row3 [0];
    }
}
```

Menunggu semua thread selesai memuat vektor ke dalam shared memory

Kernel Jacobi

Tahap 2: Menghitung Rotasi ($\cos\theta$ dan $\sin\theta$)

Proses ini hanya dikerjakan oleh thread pertama dan disimpan pada shared memory untuk mengurangi penggunaan register yang berdampak berkurangnya utilisasi prosesor.

Penggunaan satu thread untuk kode serial seperti ini membuat kernel lebih efisien: 5~6% lebih cepat daripada semua thread ikut menghitung

```
if (tid==0) {

   swap = (r1 < r2);
   if (swap) {
      S [l1] = r2;
      S [l2] = r1;
   }
   else {
      S [l1] = r1;
      S [l2] = r2;
   }

   if (|g| > δ)
      converged = false;
   if (|g| > ε) {
      w = (r2 − r1) / (2r3)

      t = sign(w) / (|w| + √(1 + w²))

      c = 1 / √(1 + t²)

      s = ct

   }
   else {
      c = 1;
      s = 0;
   }
}

__syncthreads ();
```

Implicit bubble sort→ row pivoting

Kernel Jacobi

Tahap 3: Menerapkan rotasi pada matriks A dan U.

```
for (i=0:⌈n/b⌉ − 1) {
   d1 = row1 [tid] = A [l1*n + i*b + tid];
   d2 = row2 [tid] = A [l2*n + i*b + tid];

   row1 [tid] = c*d1 – s*d2;
   row2 [tid] = s*d1 + c*d2;

   if (swap) {
      A [l2*n + i*b + tid] = row1 [tid];
      A [l1*n + i*b + tid] = row2 [tid];
   }
   else {
      A [l1*n + i*b + tid] = row1 [tid];
      A [l2*n + i*b + tid] = row2 [tid];
   }
}

for (i=0:⌈m/b⌉ − 1) {
   d1 = row1 [tid] = U [l1*n + i*b + tid];
   d2 = row2 [tid] = U [l2*n + i*b + tid];

   row1 [tid] = c*d1 – s*d2;
   row2 [tid] = s*d1 + c*d2;

   if (swap) {
      U [l2*m + i*b + tid] = row1 [tid];
      U [l1*m + i*b + tid] = row2 [tid];
   }
   else {
      U [l1*m + i*b + tid] = row1 [tid];
      U [l2*m + i*b + tid] = row2 [tid];
   }
}
```

Memuat vektor dalam kelipatan b

Dikerjakan dalam shared memory

Paralelisasi Givens-QR dalam GPU

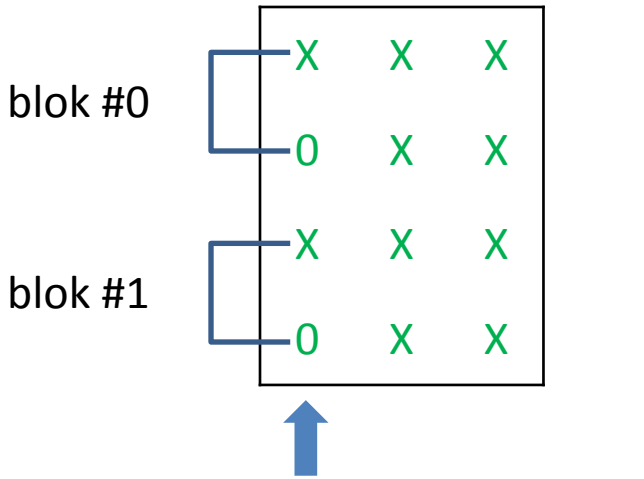
A

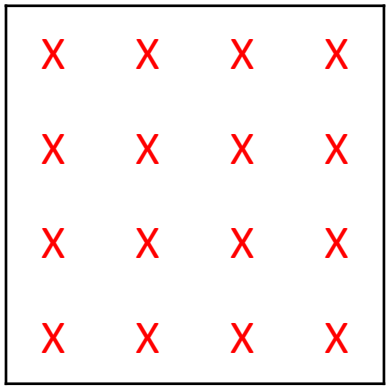
X	X	X
X	X	X
X	X	X
X	X	X

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

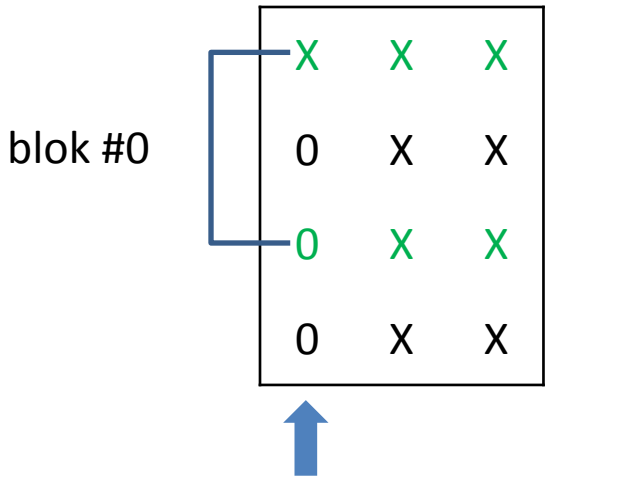
skip

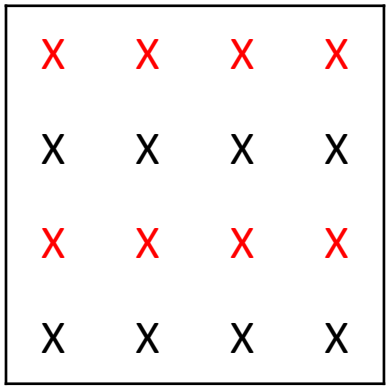
Paralelisasi Givens-QR dalam GPU



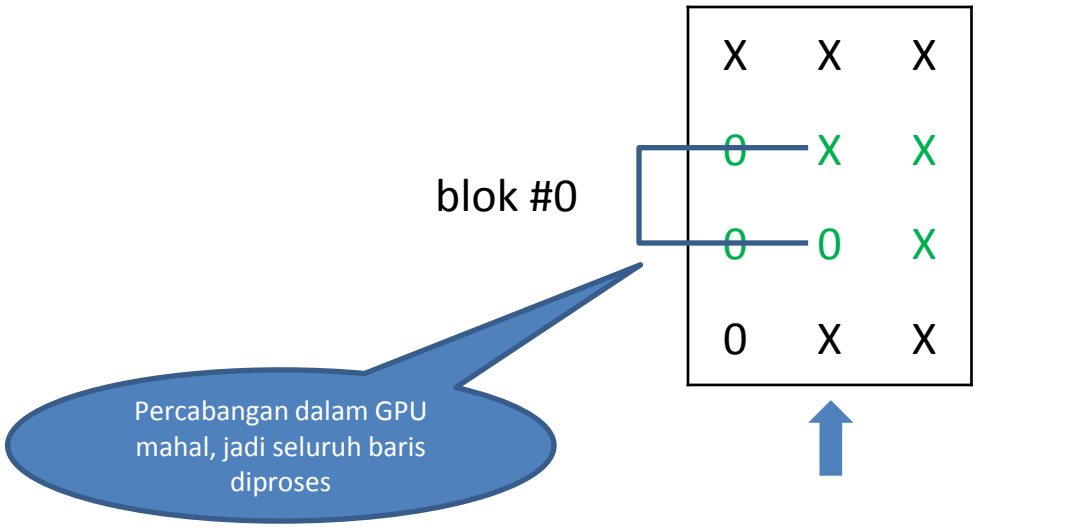


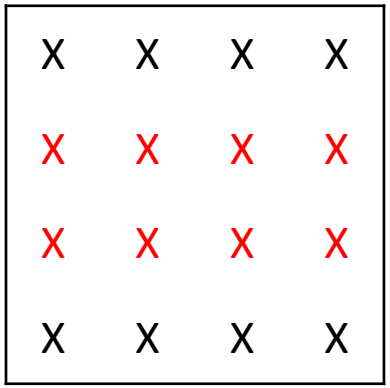
Paralelisasi Givens-QR dalam GPU



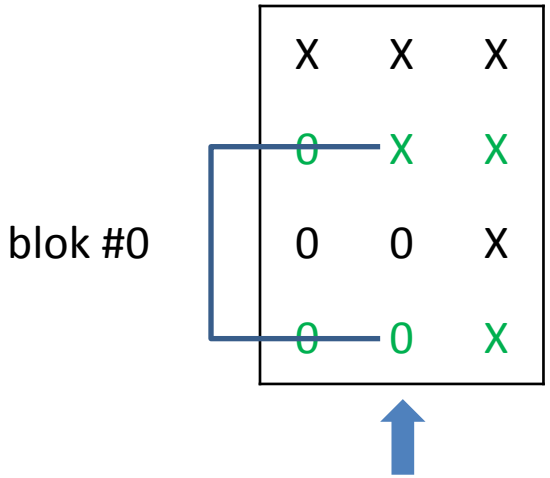


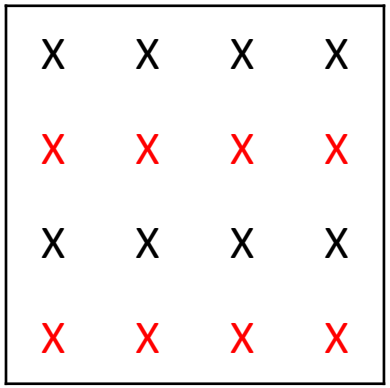
Paralelisasi Givens-QR dalam GPU



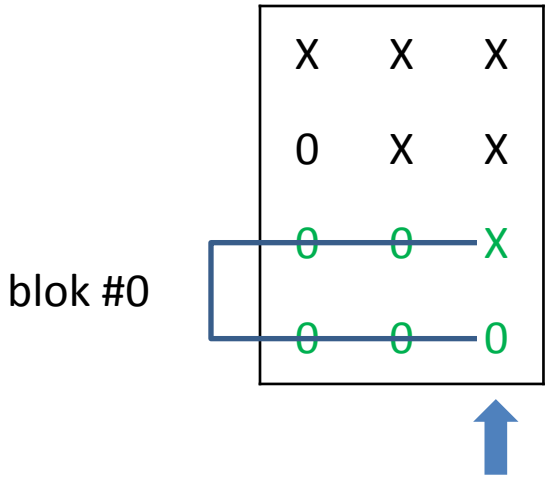


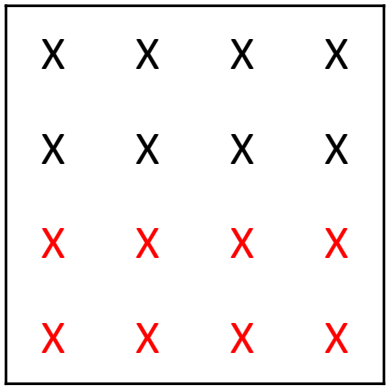
Paralelisasi Givens-QR dalam GPU





Paralelisasi Givens-QR dalam GPU





Paralelisasi Givens-QR dalam GPU

R

X	X	X
0	X	X
0	0	X
0	0	0

Q’

X	X	X	X
X	X	X	X
X	X	X	X
X	X	X	X

SVD Jacobi dengan Prekondisi QR

QR2

1. Dekomposisi QR: $A = Q_1 R$
2. Dekomposisi LQ pada R:
   1. Transpose $R \in \mathbb{R}^{n \times n}$
   2. QR pada R^T: $R^T = Q_2 L^T$
   3. Transpose Q_2, L^T
3. Jacobi SVD pada L: $L = U_1 \Sigma V_1^T$
4. Bangun V^T: $V^T = V_1^T Q_2^T$
5. Bangun U : $U = Q_1 U_1$

QR1

1. Dekomposisi QR: $A = Q_1 R$
2. Jacobi SVD pada R: $R = U_1 \Sigma V^T$
3. Bangun U: $U = Q_1 U_1$

Analisis Kompleksitas

Workload: jumlah FLOP selama rutin berjalan dalam GPU

Transfer: Jumlah transfer input dan output antara chip dan memori GPU

QR2 dan QR1 tampak tidak sensitif pada perbedaan jumlah kolom/baris.

- Jacobi
  - Memory: $M = m^2 + mn + n^2 + m$
  - Workload: $W \approx S(6m^3 + 12m^2n) + mn + n^2$
  - Transfer: $T \approx 2S(2m^3 + 3m^2n) + mn + 2n^2$
- Dekomposisi QR
  - Memory: $M = mn + m^2$
  - Workload: $W \approx 6m^2n + 3mn^2 - 3n^3$
  - Transfer: $T \approx 4m^2n + 2m^2n + 2m^2 - 2n^2$
- QR2
  - Memory: $M = 2m^2 + mn + 4n^2 + n$
  - Workload: $W \approx 6m^2n + 5mn^2 + n^3(18S + 11) - 4n^2$
  - Transfer: $T \approx 4m^2n + 4mn^2 + n^3(5S + 8) + 2m^2 + 9n^2$
- QR1
  - Memory: $M = 2m^2 + mn + 2n^2 + n$
  - Workload: $W \approx 6m^2n + 5mn^2 + 18Sn^3 - n^2$
  - Transfer: $T \approx 4m^2n + 4mn^2 + 5Sn^3 + 2m^2 + n^2$

Ujicoba dan Analisis

Pengujian Program

Percobaan: mencatat running time keempat program dan rutin SVD dari Octave

- Parameter
  - Running time
  - Throughput
  - Bandwidth
  - Speed-Up
- Tipe matriks input
  - Random
  - Hilbert
  - Image

Perangkat Pengujian

- Yang dibangun:
  - QR2
  - QR1
  - GPU Jacobi
  - CPU Jacobi
- Software
  - Cuda Toolkit 3.2
  - GCC 4.4+Linux 64 bit
  - Octave ← single CPU, *gold standard*
- Hardware
  - GPU: GeForce 9500 GT, RAM DDR2 512 MB
  - CPU: Core 2 Duo E7200, RAM DDR2 2GB

Pengujian Validitas

- Setiap babak mencakup pengujian hasil SVD
- Persyaratan validitas
  - $\max\left(\hat{U}\hat{U}^T - I\right) \leq \epsilon \rightarrow U$ ortogonal
  - $\max\left(\hat{V}\hat{V}^T - I\right) \leq \epsilon \rightarrow V$ ortogonal
  - $\max\left(\hat{U}\hat{S}\hat{V}^T - A\right) \leq \delta$

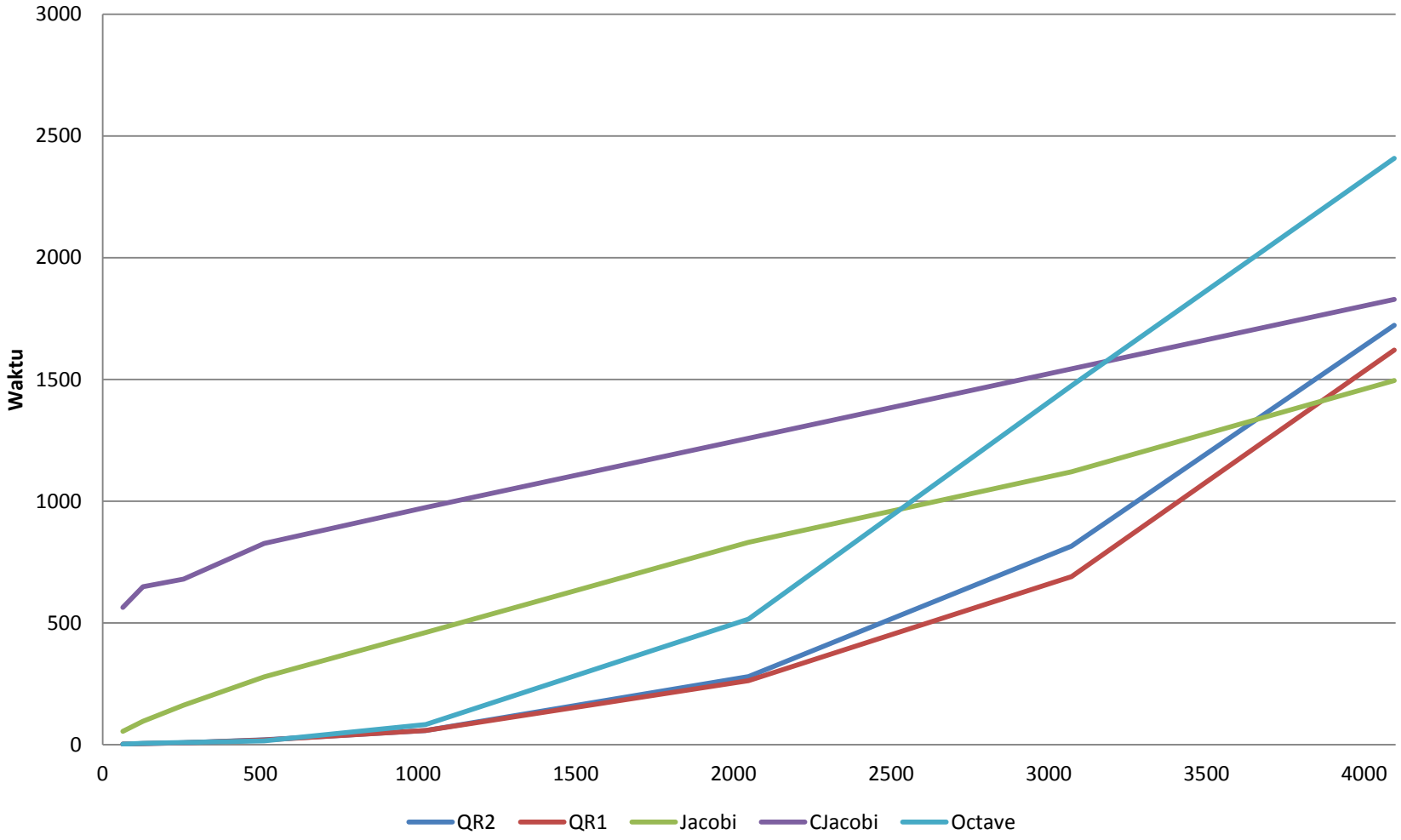
Di mana $\delta \coloneqq 10\epsilon \cdot \min(m, n)$ (HPEC Challenge 2006).

Pencatatan Waktu

- *GPU Timer* dari manajemen event CUDA
- Fungsi clock_gettime(2) untuk rutin CPU
- Tic-Toc (Octave)

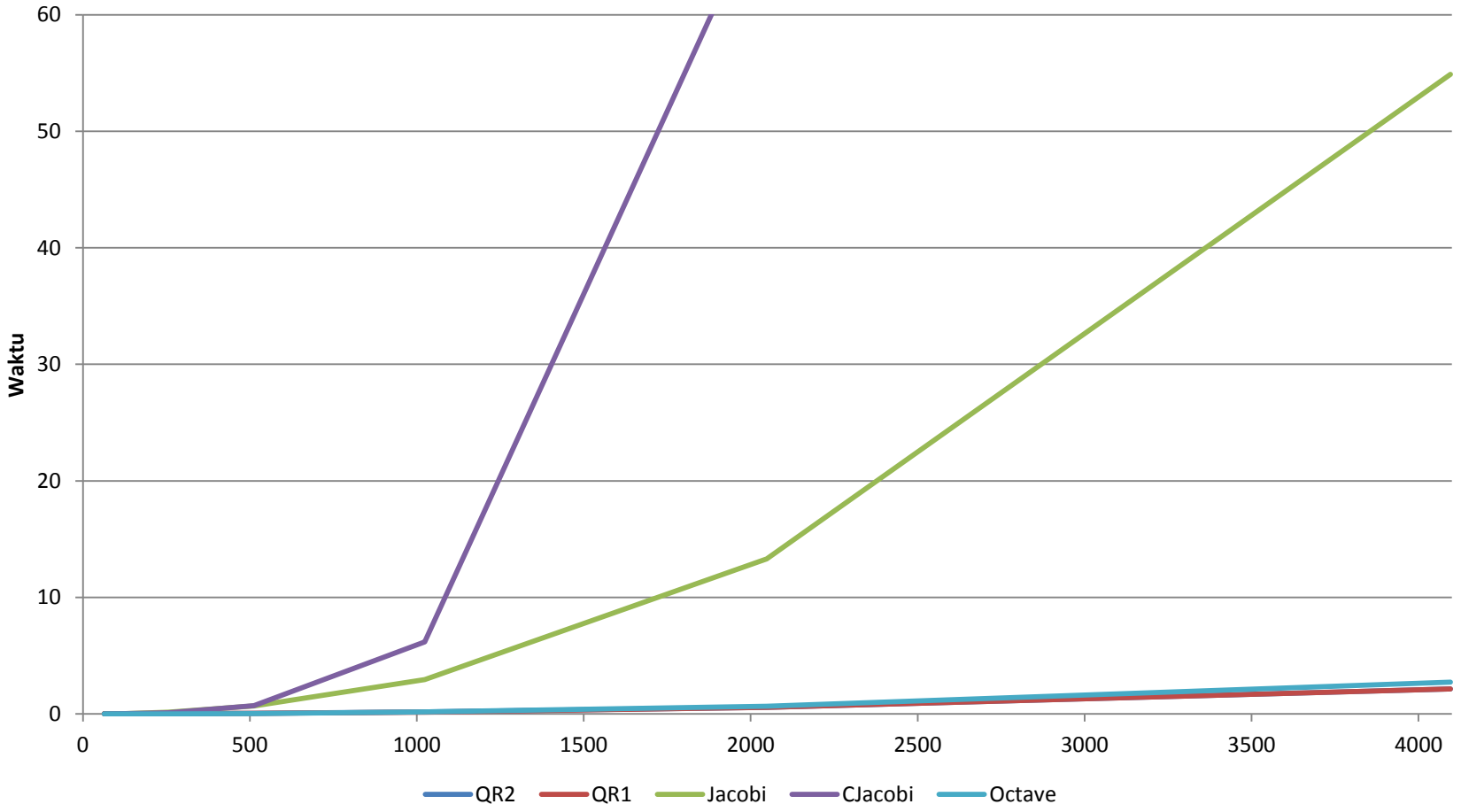
Running Time (1)

Matriks: random, Variasi: jumlah kolom, Baris: 4096



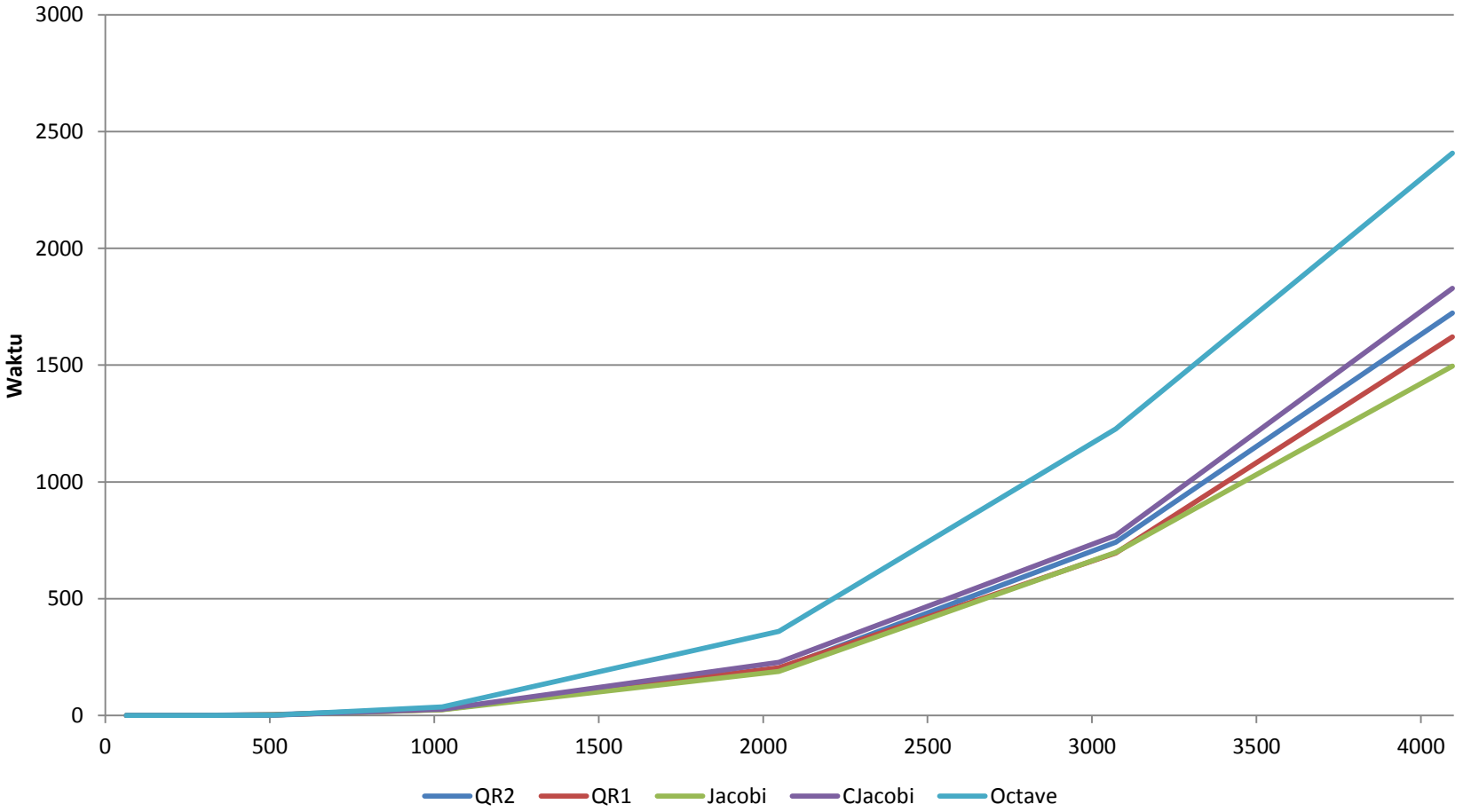
Running Time (2)

Matriks: random, Variasi: jumlah baris, Kolom: 64



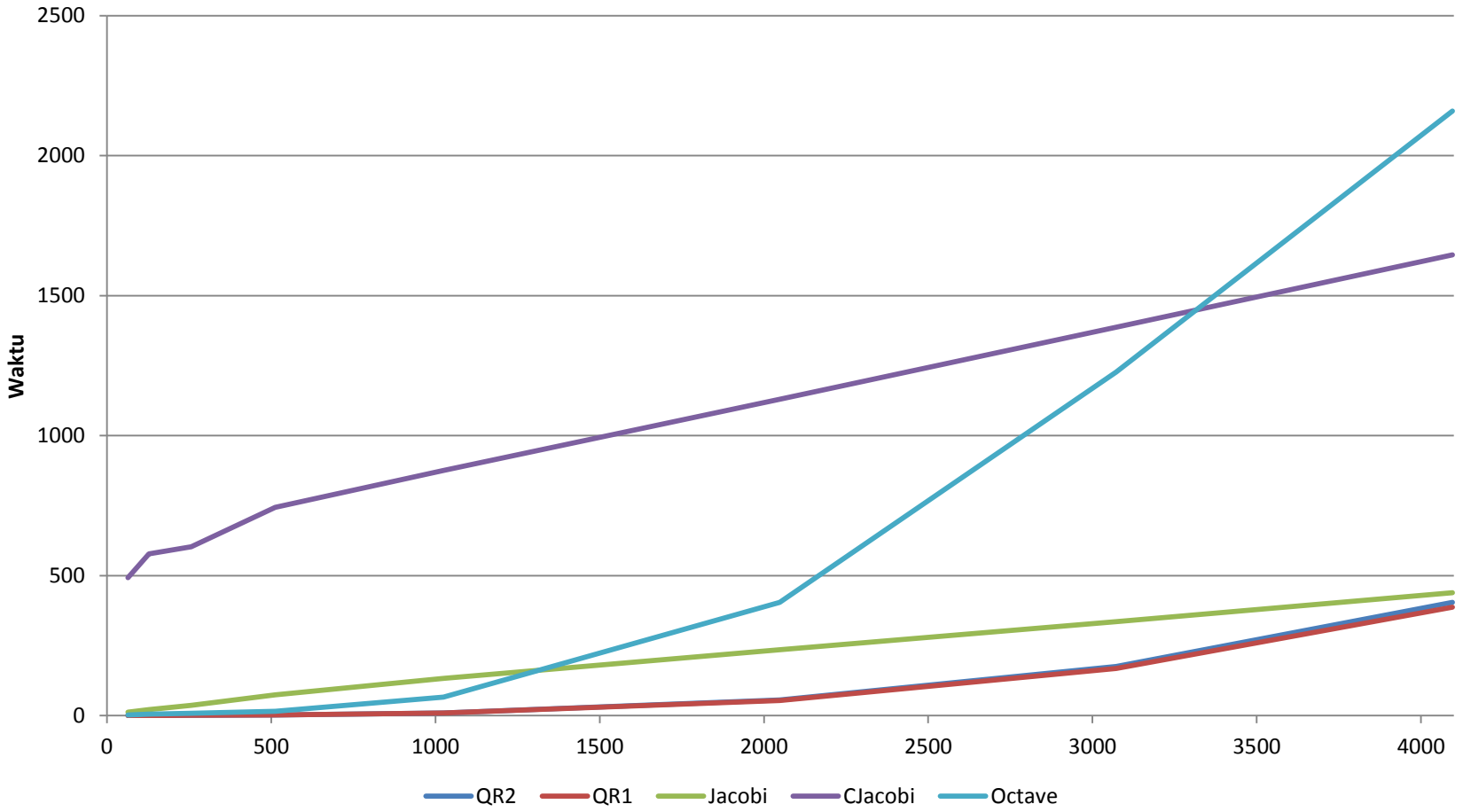
Running Time (3)

Matriks: random, Variasi: square



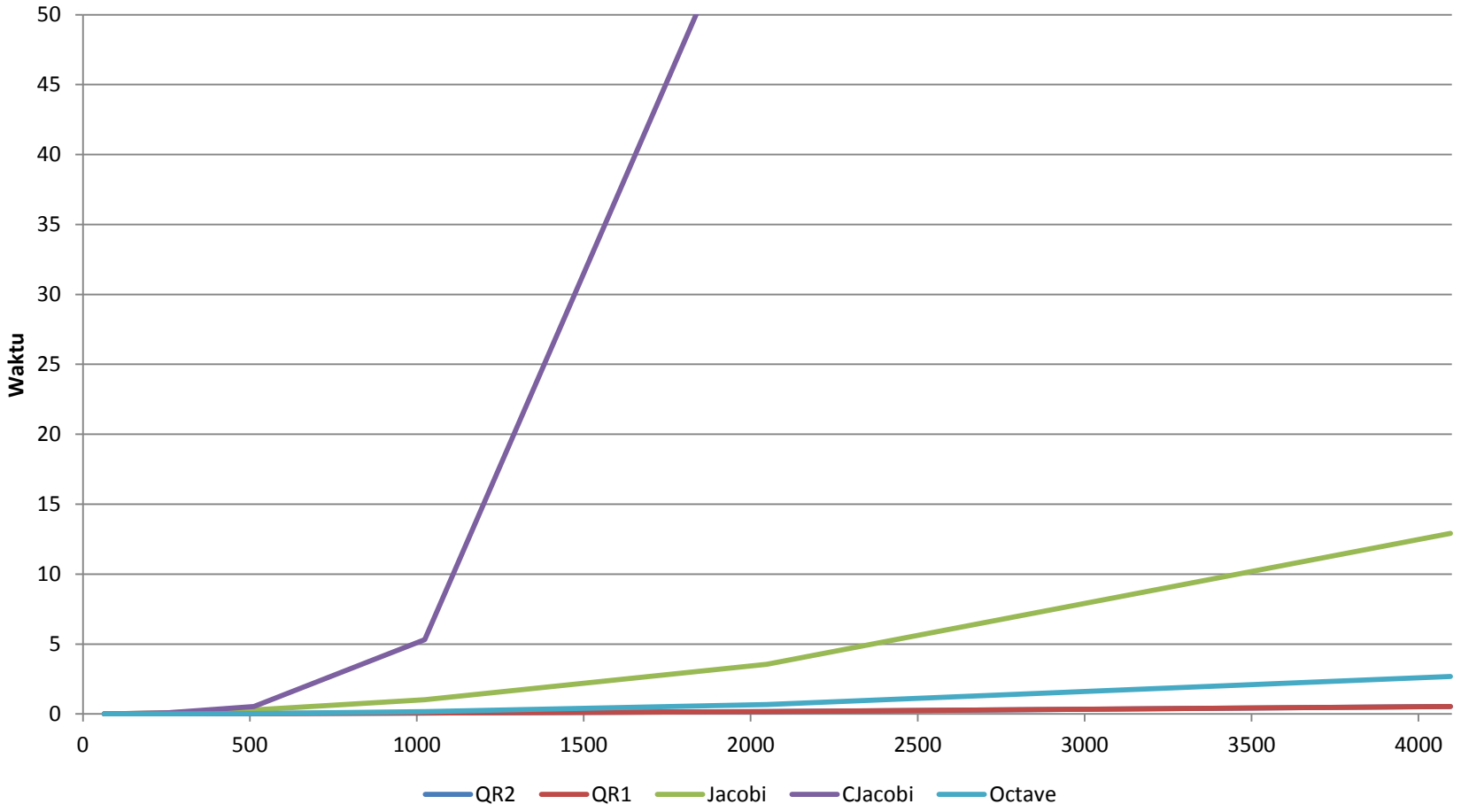
Hasil Pengujian (4)

Matriks: hilbert, Variasi: jumlah kolom, Baris: 4096

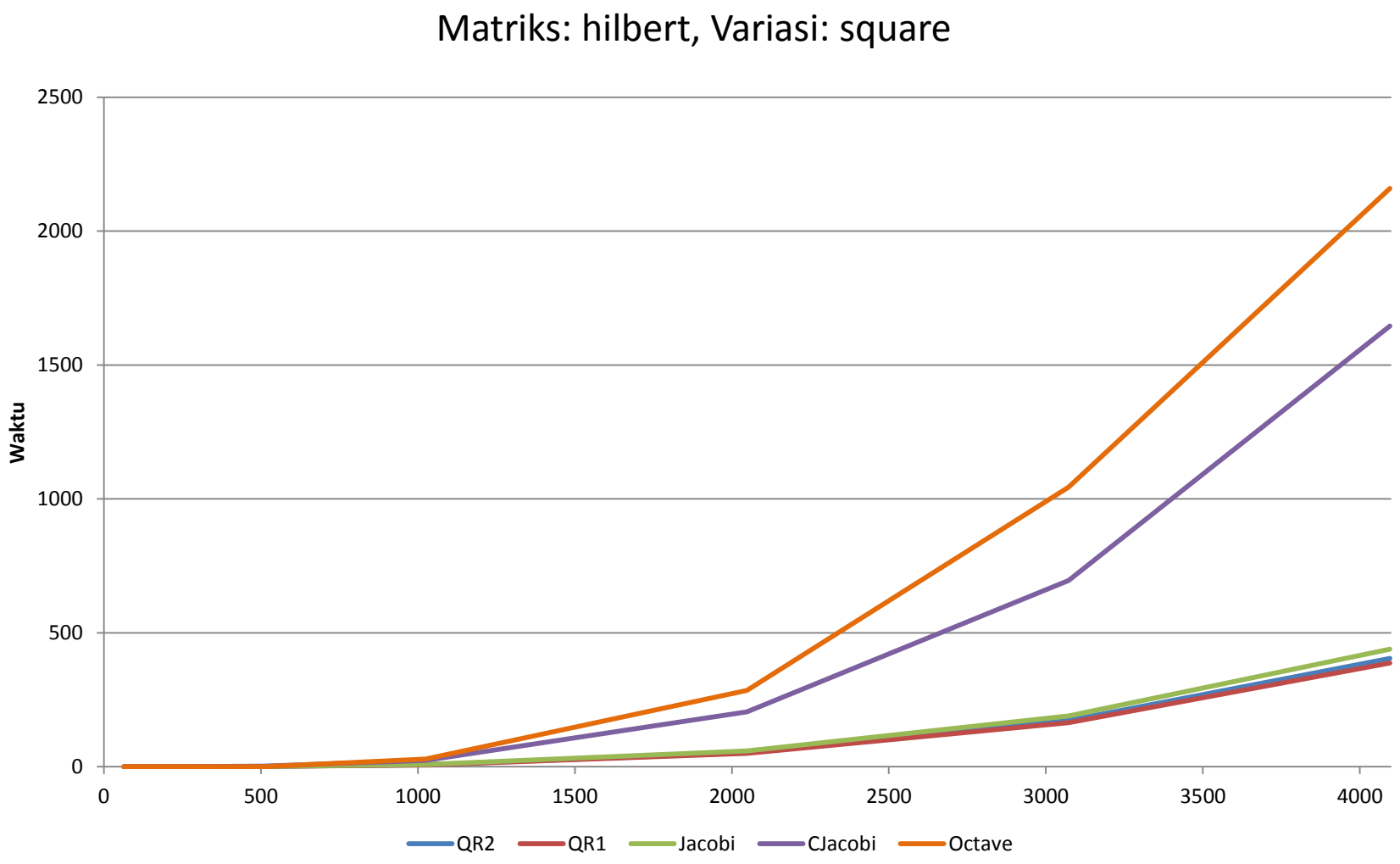


Running Time (5)

Matriks: sparse, Variasi: jumlah baris, Kolom: 64

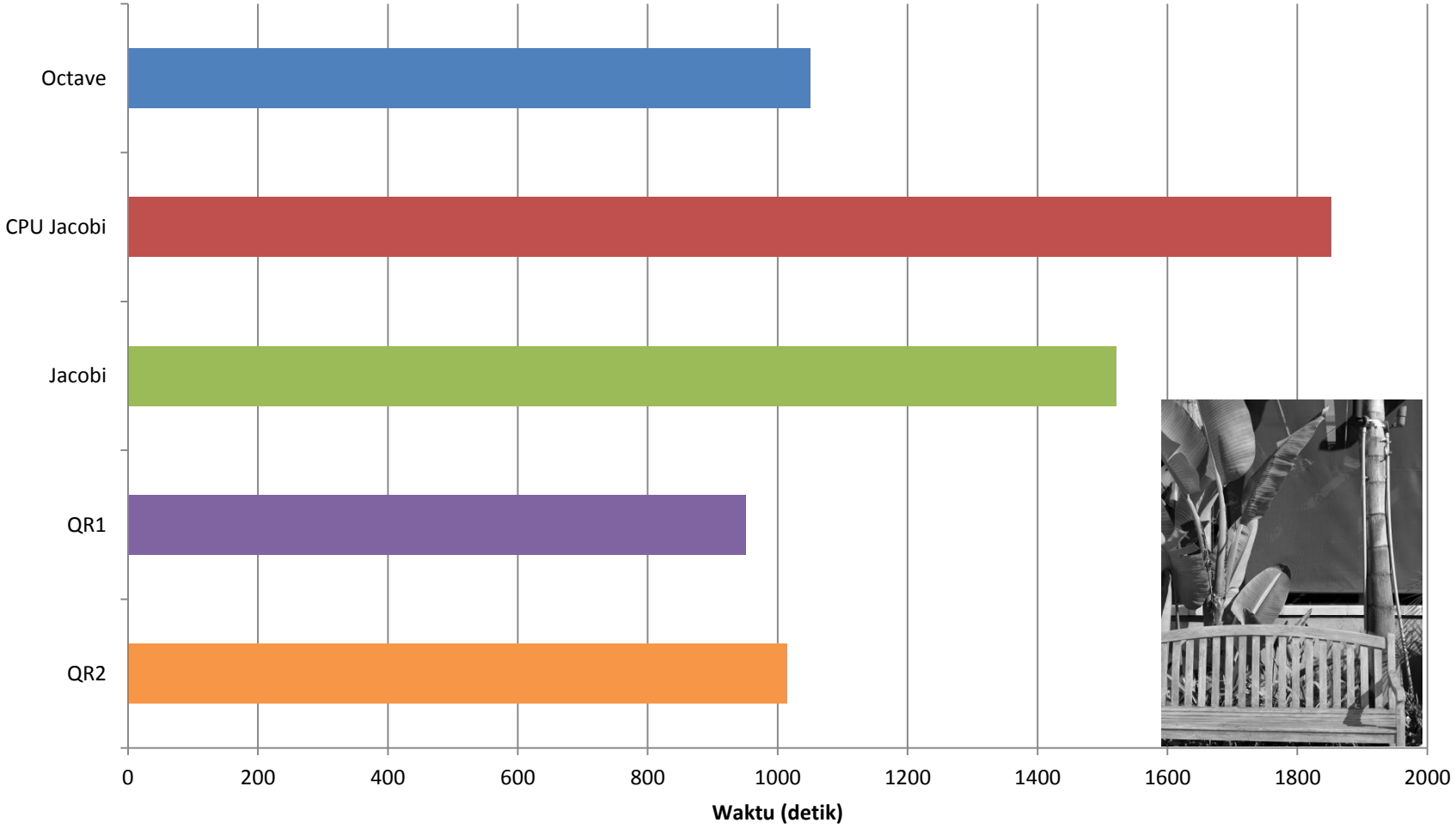


Running Time (6)



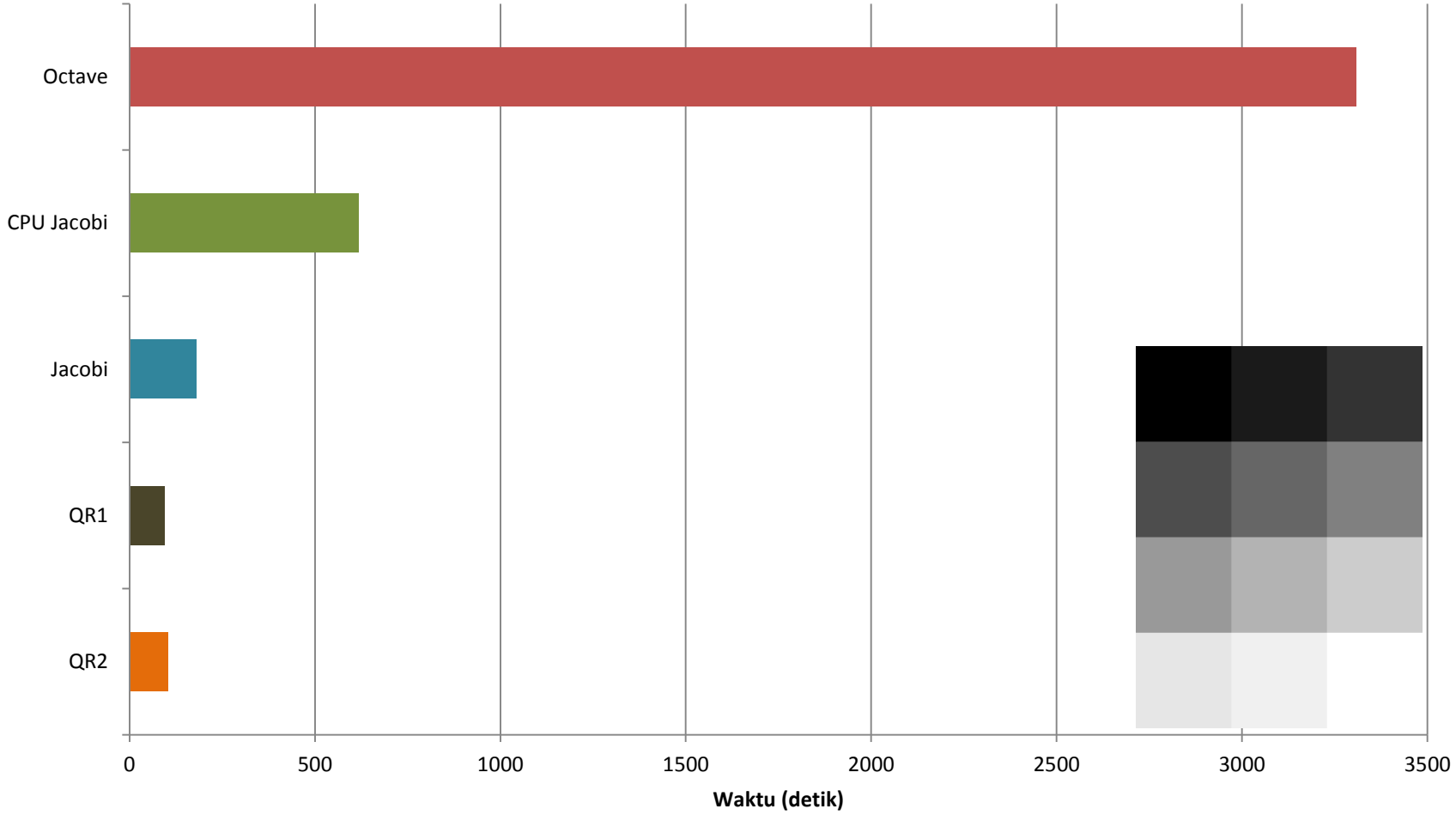
Running Time (7)

Matriks: Image 1, Ukuran: 4096×3072



Running Time (8)

Matriks: Image 2, Ukuran: 4096×3072



Pembahasan: Running Time

- Waktu komputasi sangat bergantung pada kondisi matriks sumber
- Kode GPU Jacobi secara umum lebih cepat daripada kode CPU
- Jacobi+QR pada matriks persegi lebih cepat dari Jacobi murni pada GPU dan sebaliknya pada matriks square
- Kode GPU lebih lambat untuk matriks kecil ($\leq 512 \times 512$) karena overhead komunikasi CPU←→GPU
- Seperti diprediksi, rutin Jacobi murni sangat berpengaruh terhadap jumlah sweep
- Bagaimana memprediksi jumlah sweep ?
- Octave mendapat kesulitan ketika mendapat banyak vektor tidak independen (gambar 2)

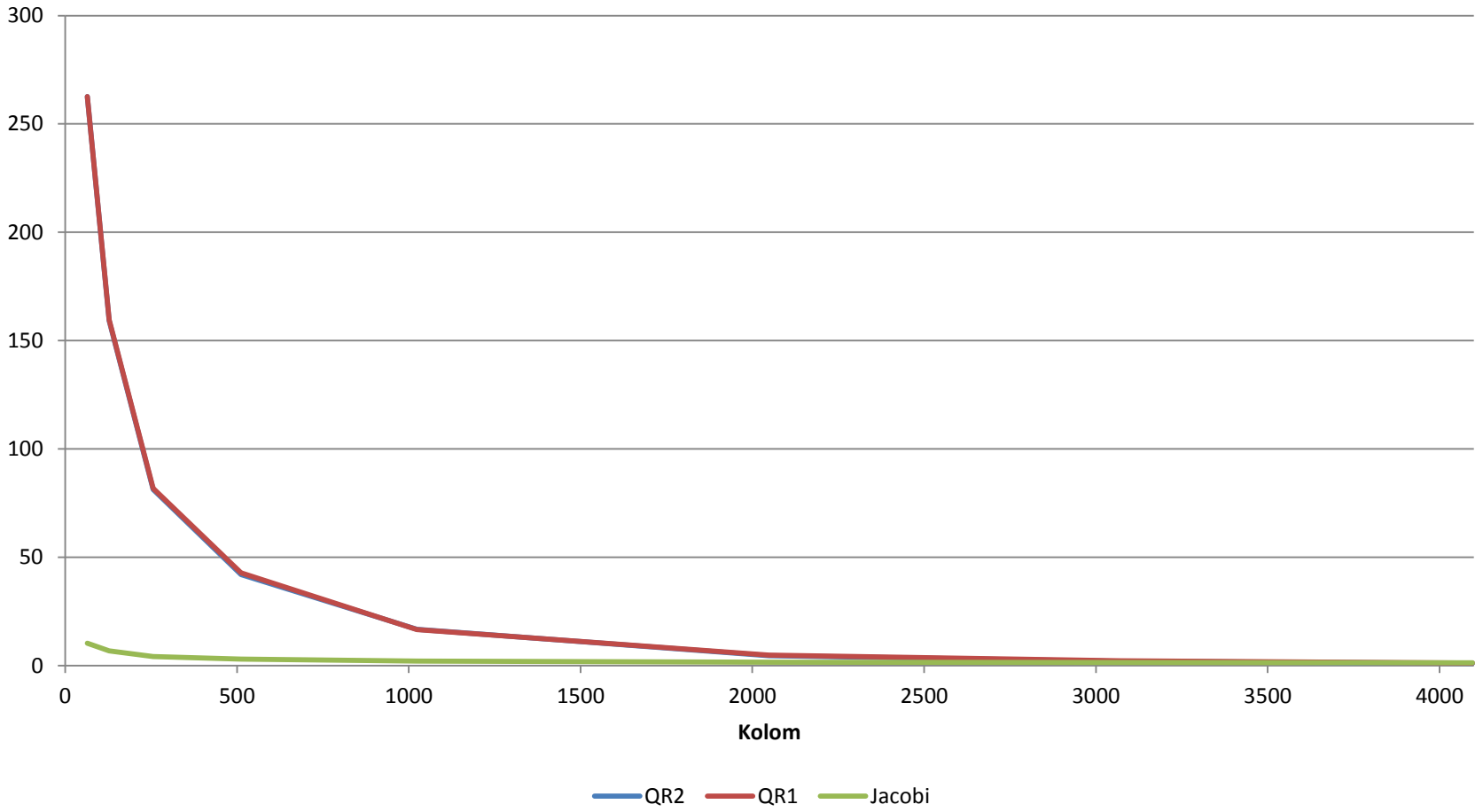
Jumlah Sweep (Matriks Random)

Ukuran	Metode			
	GPU Jacobi	CPU Jacobi	QR2	QR1
64×64	8	8	8	8
256×64	9	9	7	7
2048×2048	10	10	9	10
4096×2048	11	10	10	10
4096×3072	10	10	10	9
4096×4096	10	10	10	10

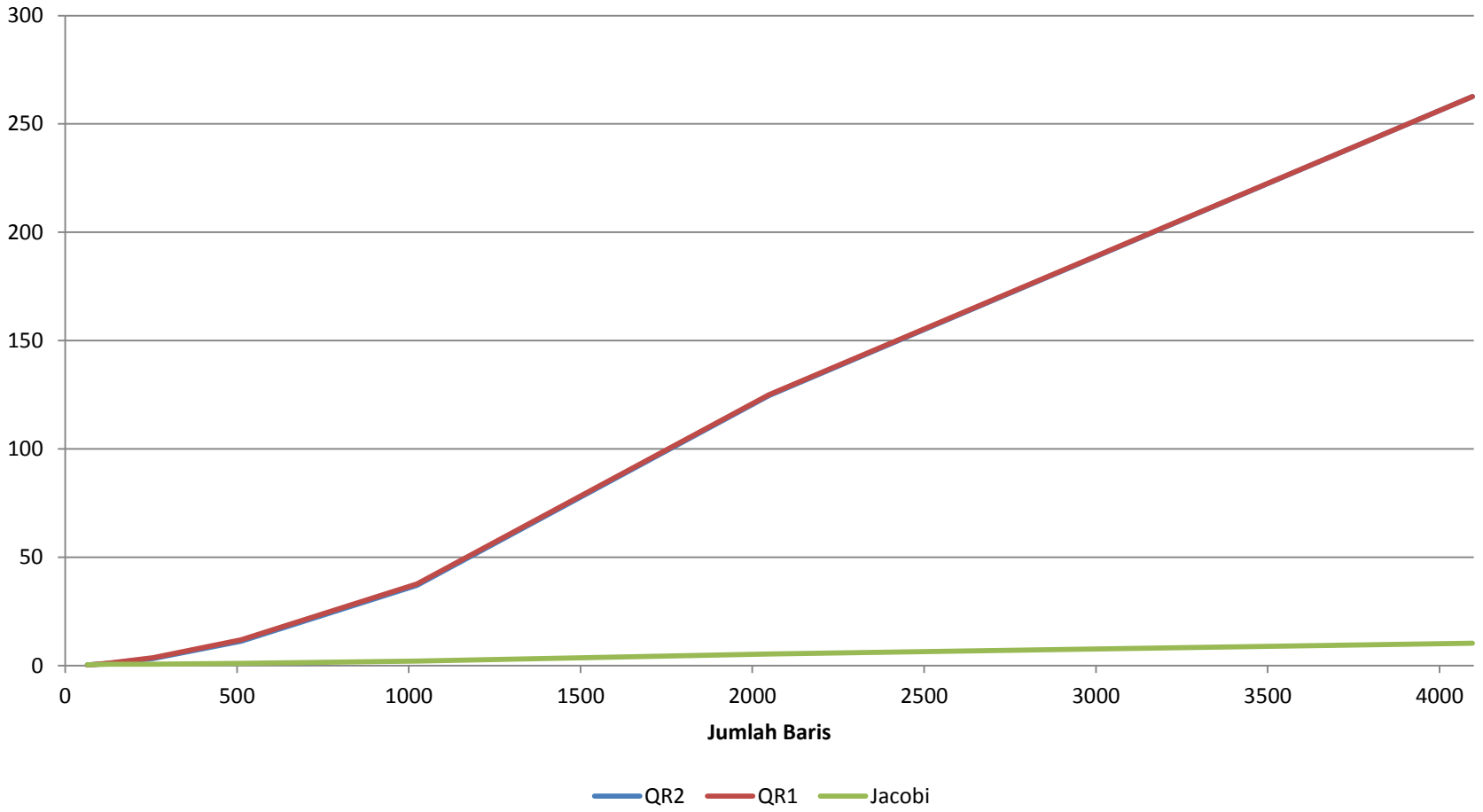
Jumlah Sweep (Matriks Hilbert)

Ukuran	Metode			
	GPU Jacobi	CPU Jacobi	QR2	QR1
64×64	6	6	6	6
256×64	7	7	7	7
2048×2048	9	9	8	8
4096×2048	9	9	8	8
4096×3072	9	9	8	8
4096×4096	9	9	8	8

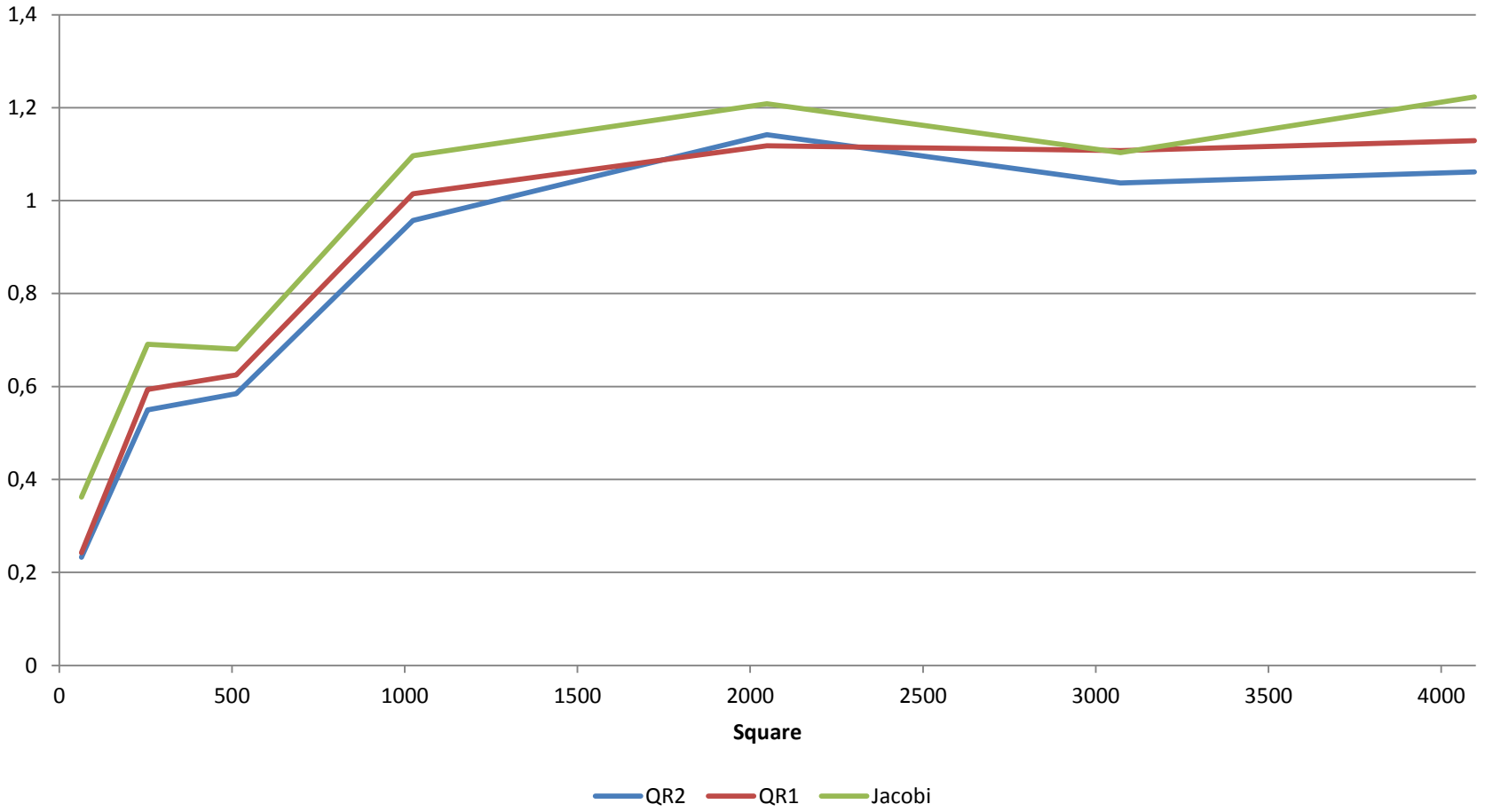
Speed-Up (1)


300
250
200
150
100
50
0
0
500
1000
1500
2000
2500
3000
3500
4000
Kolom
QR2
QR1
Jacobi

Speed-Up (2)


300
250
200
150
100
50
0
0
500
1000
1500
2000
2500
3000
3500
4000
Jumlah Baris
QR2
QR1
Jacobi

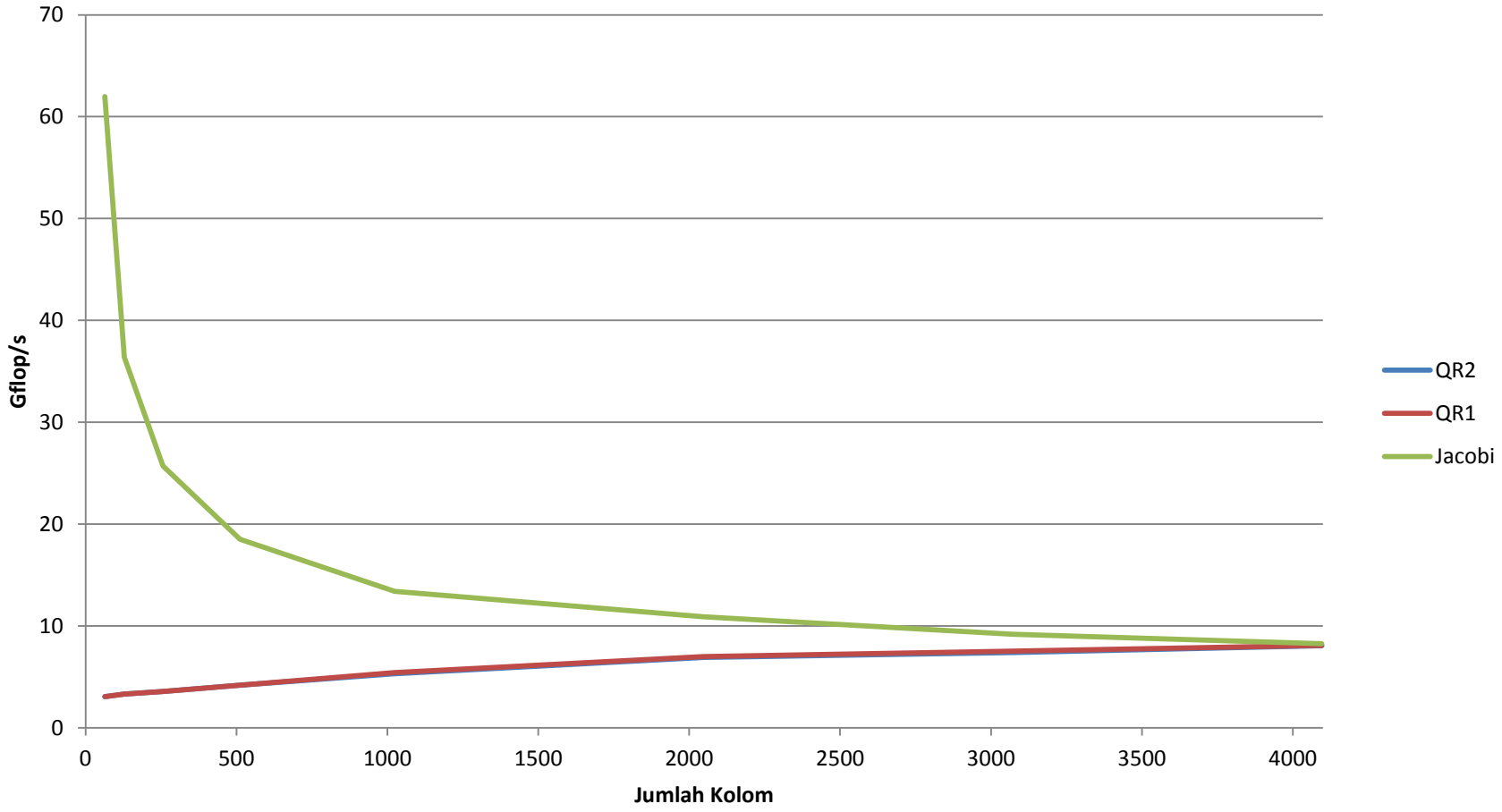
Speed-Up (3)


1,4
1,2
1
0,8
0,6
0,4
0,2
0
0
500
1000
1500
2000
2500
3000
3500
4000
Square
QR2
QR1
Jacobi

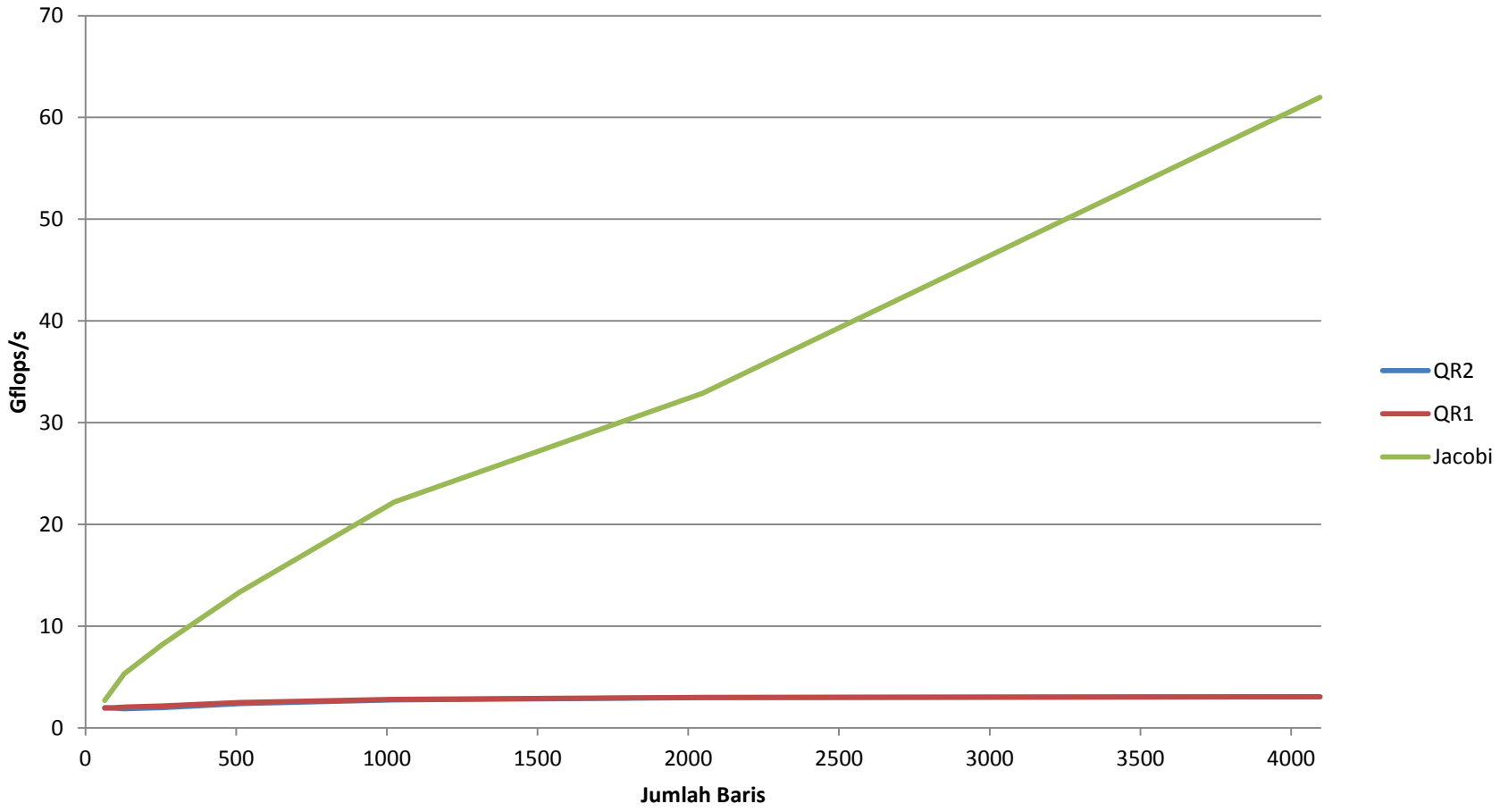
Pembahasan: Speed-Up

- Kode GPU memiliki speed-up rendah untuk jumlah baris kecil ($m \leq 256$)→overhead sinkronisasi CPU-GPU sangat besar
- Pada matriks square berukuran besar, kode Jacobi pada GPU tidak unggul signifikan

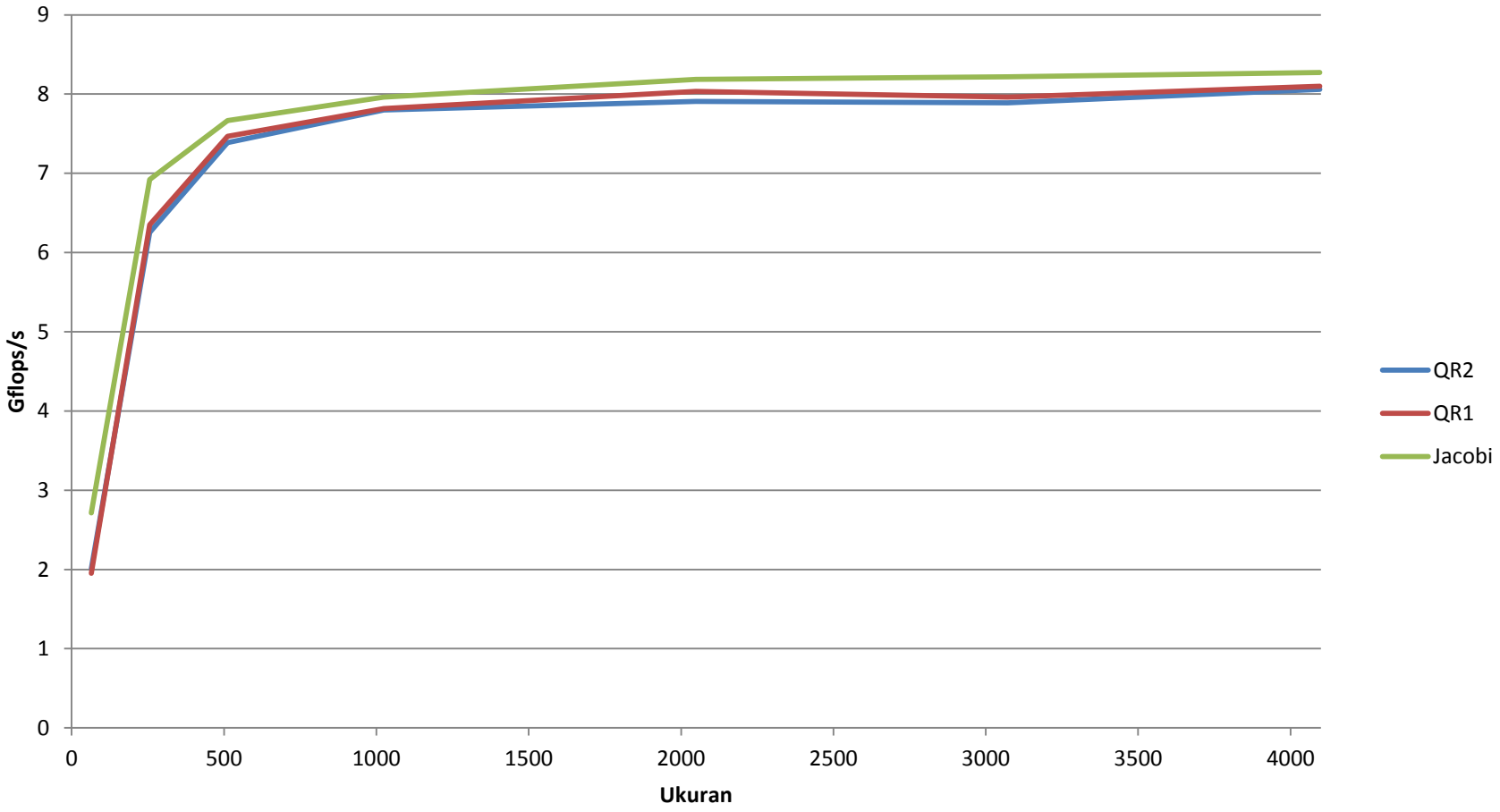
Throughput (1)



Throughput (2)



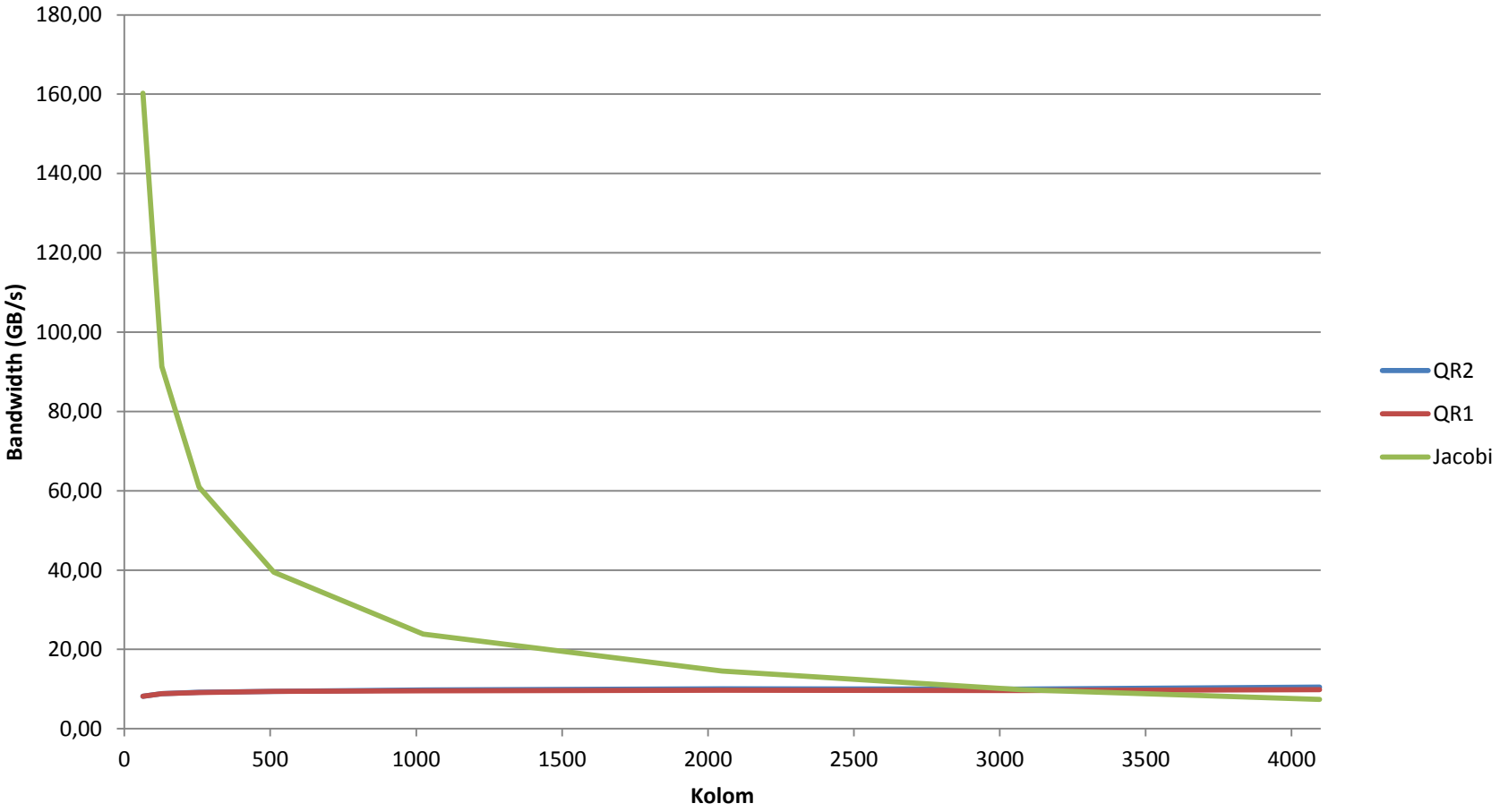
Throughput (3)



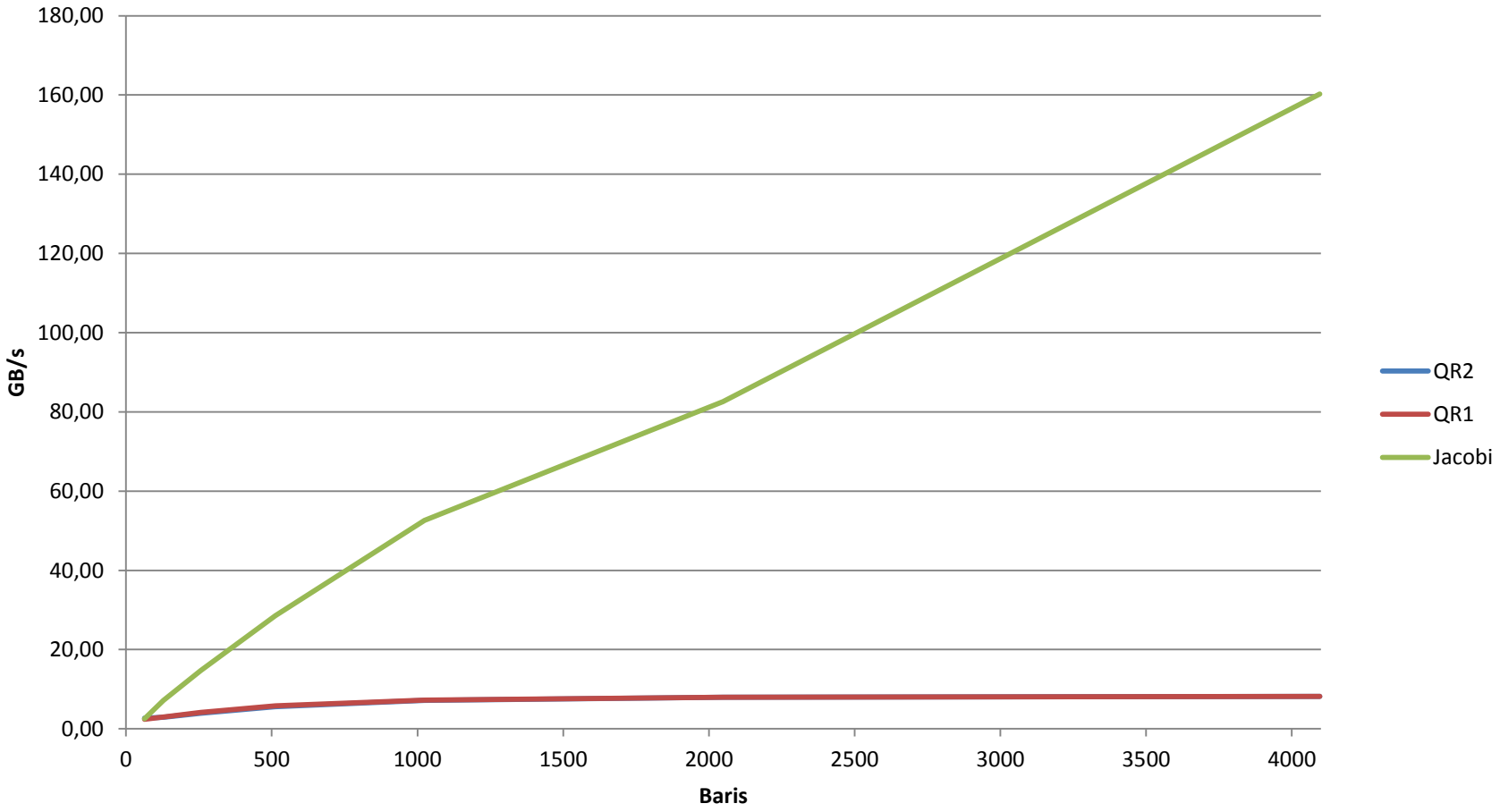
Pembahasan: Throughput

- Perbedaan karakteristik dalam throughput QR2 dan QR1 tidak signifikan, namun QR2 membutuhkan memori lebih besar
- Ketiga algoritma ini masih berada di bawah batas throughput GPU (134 Gflop/s)
- Perlu optimalisasi kode floating point dan pola akses memori

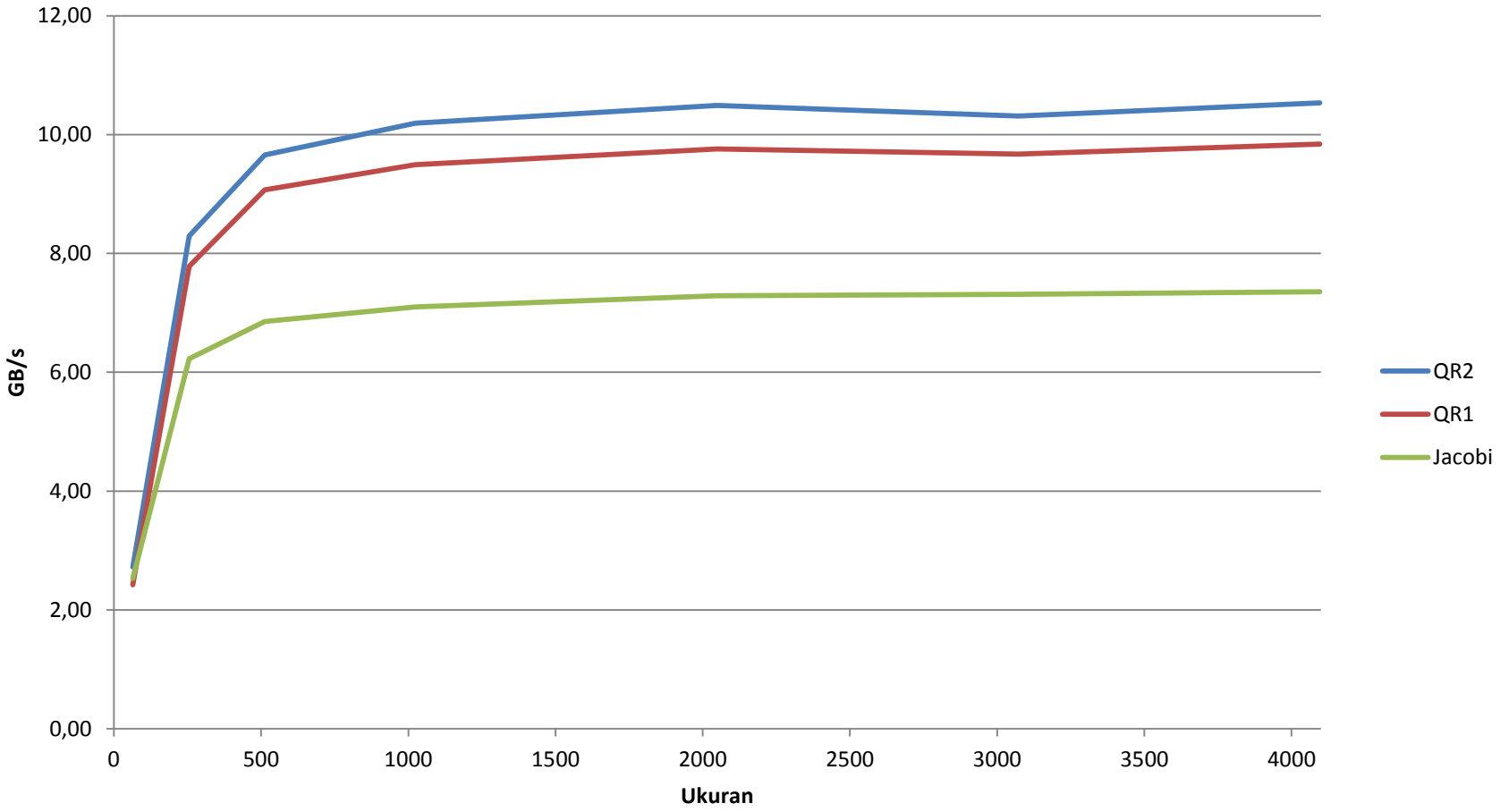
Bandwidth (1)


180,00
160,00
140,00
120,00
100,00
80,00
60,00
40,00
20,00
0,00
Bandwidth (GB/s)
0
500
1000
1500
2000
2500
3000
3500
4000
Kolom
QR2
QR1
Jacobi

Bandwidth (2)


180,00
160,00
140,00
120,00
100,00
80,00
60,00
40,00
20,00
0,00
GB/s
0
500
1000
1500
2000
2500
3000
3500
4000
Baris
QR2
QR1
Jacobi

Bandwidth (3)


12,00
10,00
8,00
6,00
4,00
2,00
0,00
GB/s
0
500
1000
1500
2000
2500
3000
3500
4000
Ukuran
QR2
QR1
Jacobi

Pembahasan (Bandwidth)

- Anomali pada Jacobi berukuran “kurus”: bandwidth maksimal GPU adalah 32 GB/s.
- Dugaan: efek *cache memory* dalam prosesor namun transparan dari software
- Perlu perubahan kode untuk eksploitasi cache
- Perlu perubahan kode untuk memperbaiki pola akses memori

Hasil Validasi

- Metode Jacobi dapat menghitung SVD secara akurat, sesuai dengan akurasi mesin (presisi tunggal)
- Pengecualian: kasus rank-deficient (contoh: gambar 2)
- Nilai-nilai singular berhasil dihitung, namun matriks U dan/atau V^T mengandung NaN

Kesimpulan dan Saran

Kesimpulan

- Metode Jacobi SVD dalam GPU dapat mengungguli metode sebelumnya yang berbasis bidiagonalisasi, terutama pada matriks berukuran besar
- Penggunaan urutan ortogonalisasi paralel menurut anti-diagonal mampu mencapai konvergensi
- Perlakuan prekondisi dengan QR mampu mengurangi waktu untuk metode Jacobi pada keadaan $\mathrm{m} \gg n$
- Metode Jacobi belum mampu menangani kasus rank-deficient

Saran

Masih terdapat ruang untuk perbaikan metode Jacobi dalam GPU:

- Eksploitasi bentuk matriks segitiga hasil prekondisi dengan QR/LQ: metode Kogbetliantz
- Penanganan kasus *rank-deficient*
- Penanganan kasus $m \leq n$. Dapat dilakukan dengan lebih dulu transpose matriks *in-place*.

Terima Kasih