

# *Oblikovanje i analiza algoritama*

## *4. predavanje*

Saša Singer

`singer@math.hr`

`web.math.pmf.unizg.hr/~singer`

PMF – Matematički odsjek, Zagreb

# Sadržaj predavanja

- Složenost u praksi — eksperimenti (nastavak):
  - Množenje matrica reda  $n$ .
  - Blokovsko množenje matrica reda  $n$ .

# Informacije — web stranica

Moja web stranica za Oblikovanje i analizu algoritama je

<https://web.math.pmf.unizg.hr/~singer/oaa/>

ili, skraćeno

<https://web.math.hr/~singer/oaa/>

Kopija je na adresi

<http://degiorgi.math.hr/~singer/oaa/>

Službena web stranica za Oblikovanje i analizu algoritama je

<https://web.math.pmf.unizg.hr/nastava/oaa/>

# Informacije — kolokviji

Oblikovanje i analiza algoritama je u kolokvijskom razredu **B1**.

Službeni termini svih kolokvija su:

- Prvi kolokvij: srijeda, 23. 11. 2016., u 9 sati.
- Drugi kolokvij: srijeda, 1. 2. 2017., u 9 sati.
- Popravni kolokvij: srijeda, 15. 2. 2017., u 9 sati.

Uputa: “izbjegnite” popravni — obavite to ranije!

# Množenje matrica

# Množenje matrica

**Problem:** Zadan je prirodni broj  $n \in \mathbb{N}$  i 3 matrice  $A$ ,  $B$  i  $C$ , reda  $n$ . Treba izračunati izraz

$$C := C + A * B.$$

Akumulacija (“nazbrajavanje”) produkta  $A * B$  u matrici  $C$

• standardni je oblik BLAS-3 rutine **xGEMM** za množenje matrica,

tj. baš ova operacija se često koristi u praksi.

Usput, to će opet

• “prevariti” optimizaciju kompilera, kod višestrukog ponavljanja eksperimenta.

# Množenje matrica — formula

“Matematička” realizacija **matrične** operacije

$$C := C + A * B$$

po **elementima** je trivijalna:

$$c_{ij} := c_{ij} + \sum_{k=1}^n a_{ik} \cdot b_{kj},$$

za sve indekse

$$i = 1, \dots, n, \quad j = 1, \dots, n.$$

Dakle, “programski” — treba “zavrtiti” **tri** petlje.

# Množenje matrica — potprogram

```
subroutine mulijk (lda, n, a, b, c)
c
c Matrix multiply
c  $C(n, n) = C(n, n) + A(n, n) * B(n, n)$ .
c
c   implicit none
c
c   integer lda, n
c   double precision a(lda, lda), b(lda, lda),
c   $               c(lda, lda)
c
c   integer i, j, k, nn
```



# Množenje matrica — potprogram (nastavak)

```
c
c   IJK loop, inner
c
      nn = n
      do 30, i = 1, nn
        do 20, j = 1, nn
          do 10, k = 1, nn
            c(i, j) = c(i, j) + a(i, k) * b(k, j)
          10      continue
        20      continue
      30      continue
c
      return
      end
```

## Permutacija petlji

Ovu varijantu algoritma zovemo **ijk** — opet po **poretku** (indeksa) petlji, **izvana** prema **unutra**.

Sve **tri** petlje možemo **permutirati**, tj. napisati ih u **bilo kojem** poretku. Na taj način dobivamo ukupno **6** varijanti algoritma, koje zovemo leksikografskim redom:

- **ijk**,
- **ikj**,
- **jik**,
- **jki**,
- **kij**,
- **kji**.

# Broj operacija

U svakom prolazu kroz unutarnju petlju imamo dvije operacije:

- množenje matričnih elemenata  $a_{ik} \cdot b_{kj}$ ,
- zbrajanje tog produkta s  $c_{ij}$ .

Sve tri petlje imaju (svaka) točno  $n$  prolaza.

Ukupan broj operacija u svim varijantama algoritma je:

$$F(n) = 2n^3.$$

Broj ponavljanja  $N(n)$  izabran je tako da dobijemo približno konstantno trajanje “okolne” petlje (s ponavljanjem) kojoj mjerimo vrijeme, sve dok  $N(n)$  ne padne na 1, za  $n = 450$ .

# Boje na grafovima

Legenda za čitanje grafova:

- petlja **ijk** — zeleno, rang 3;
- petlja **ikj** — narančasta, rang 5;
- petlja **jik** — žuta, rang 4;
- petlja **jki** — ljubičasta, rang 1;
- petlja **kij** — crveno, rang 6;
- petlja **kji** — plavo, rang 2.

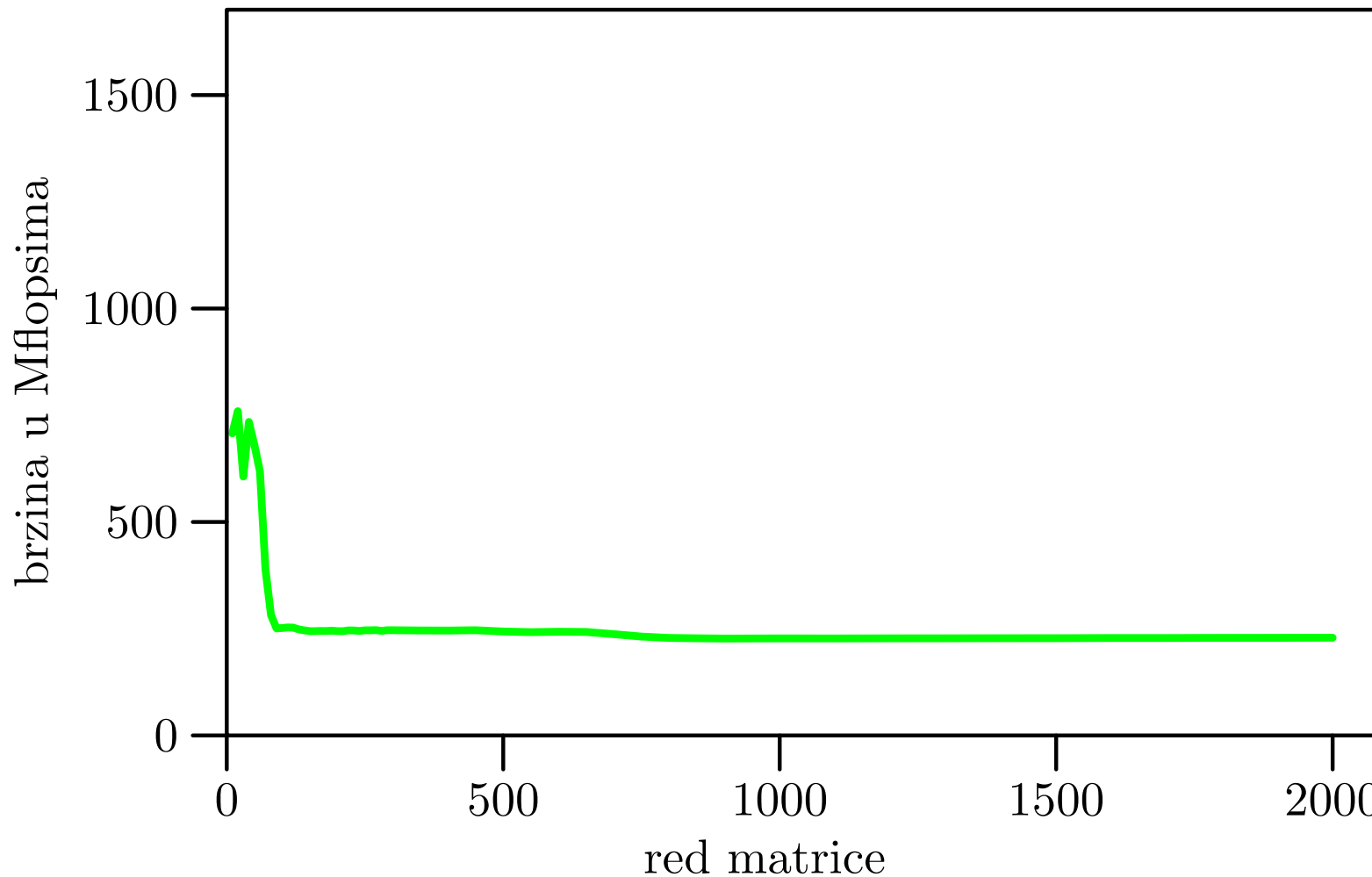
# BabyBlue, CVF, normal

## Compaq Visual Fortran:

- normalna optimizacija:
  - prvo 6 pojedinačnih slika, leksikografskim redom, po petljama,
  - a zatim, zajednički graf za svih 6 petlji.
- fast optimizacija:
  - permutira petlje, tako da svih 6 petlji daje gotovo istu brzinu.
- Usporedba:
  - najbrže petlje jki u fast optimizaciji i
  - MKL-ovog algoritma DGEMM.

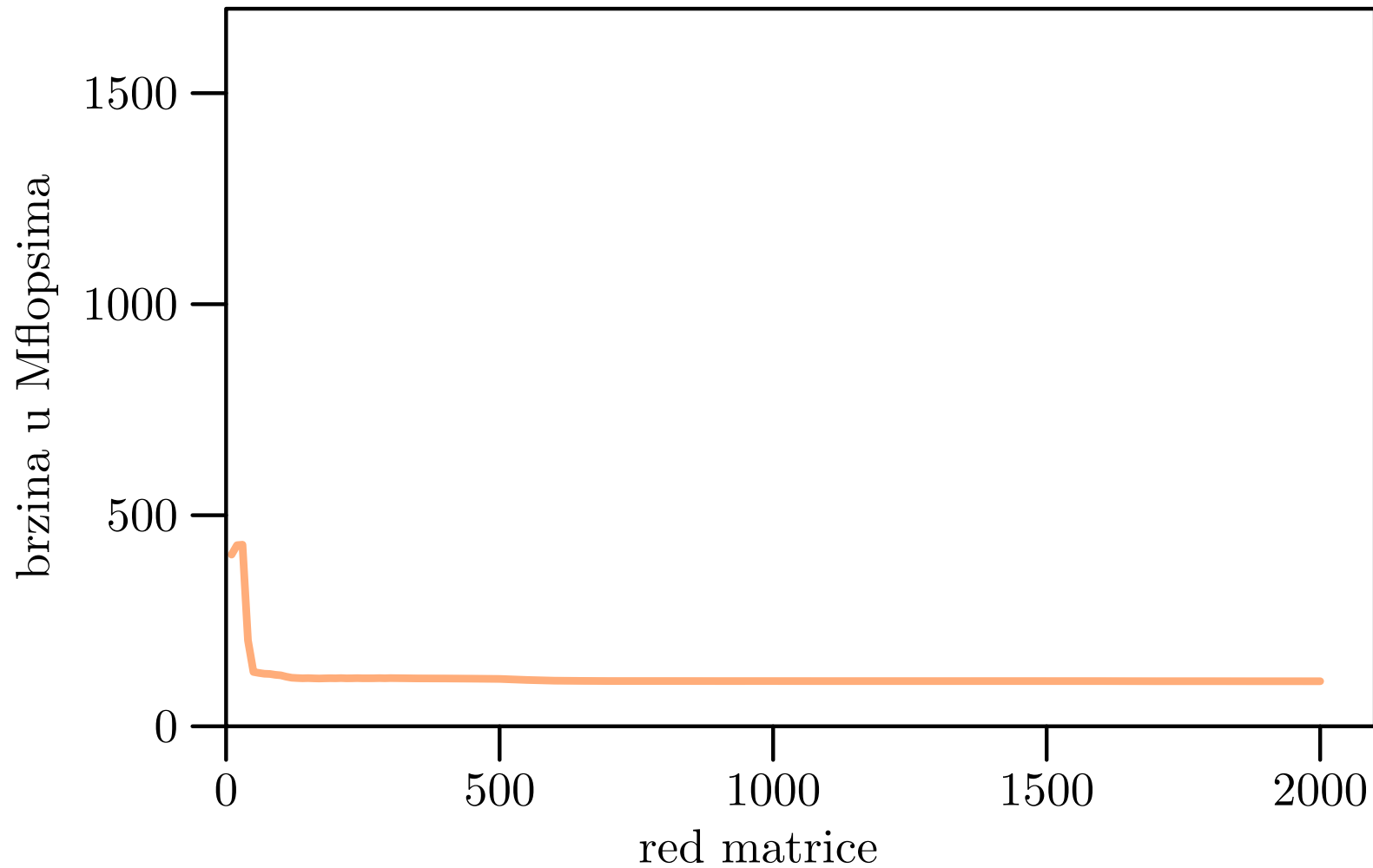
# BabyBlue, CVF, normal — ijk

Pentium 4/660, 3.6 GHz, CVF, normal – Množenje matrica ijk



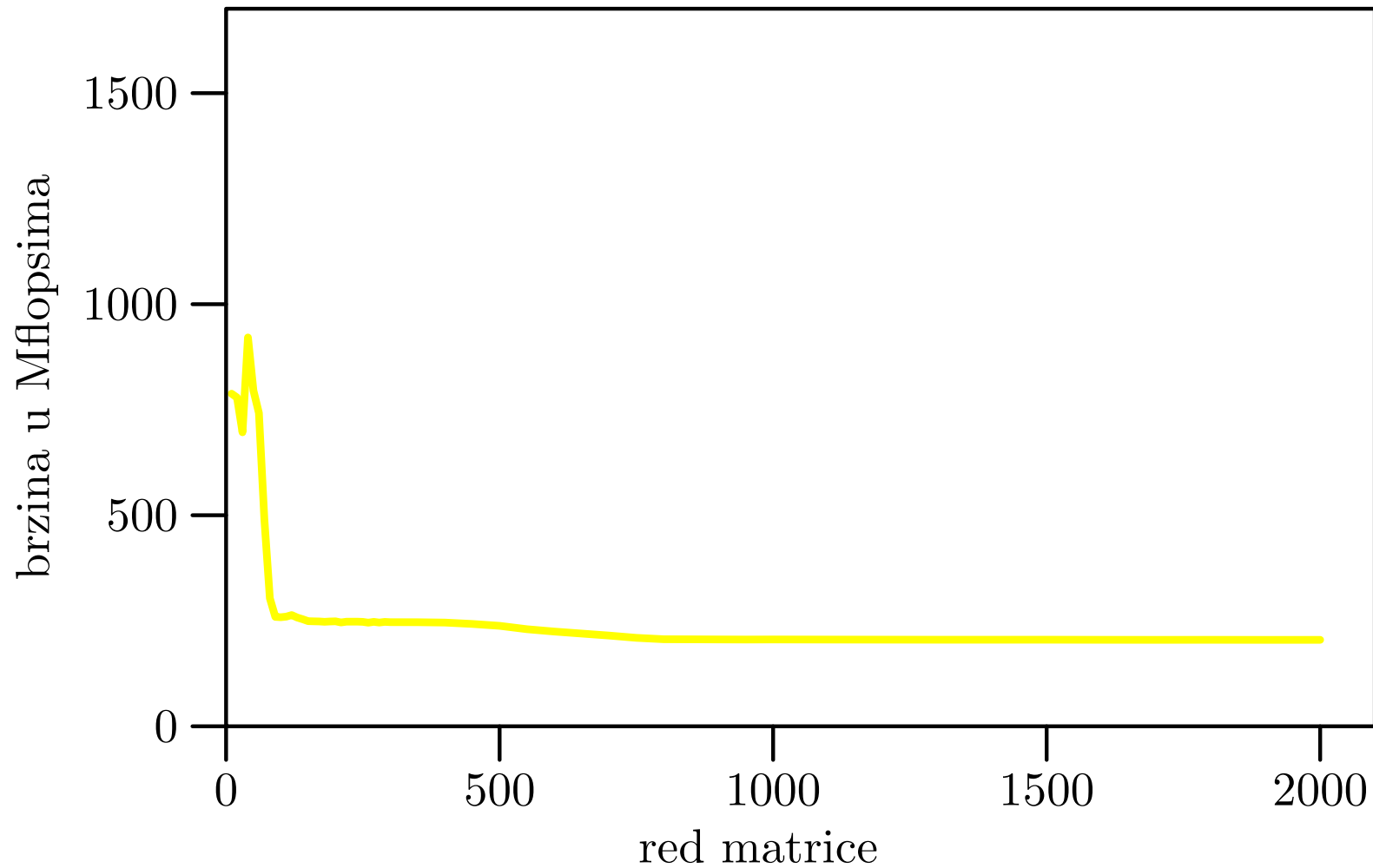
# BabyBlue, CVF, normal — ikj

Pentium 4/660, 3.6 GHz, CVF, normal – Množenje matrica ikj



# BabyBlue, CVF, normal — jik

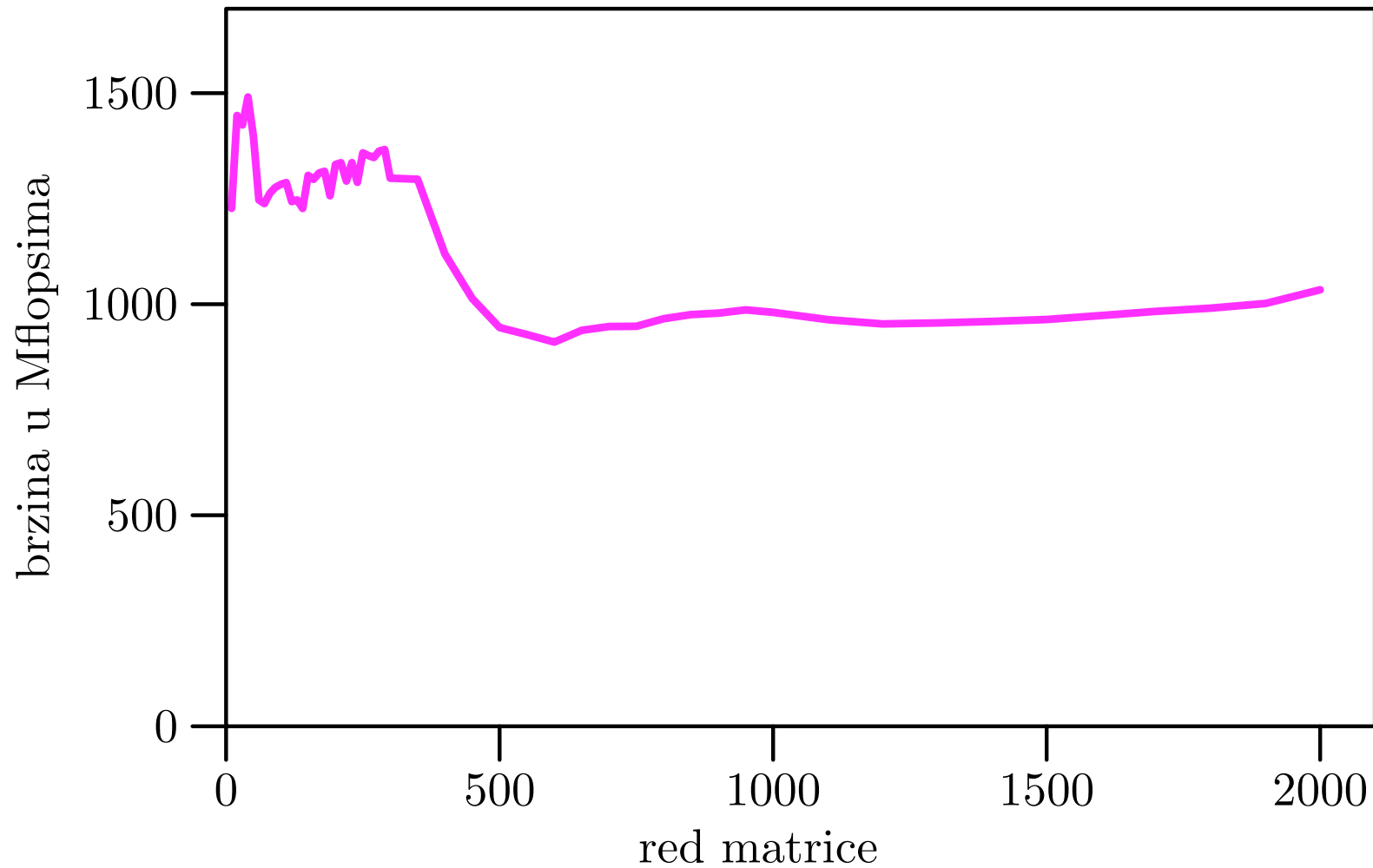
Pentium 4/660, 3.6 GHz, CVF, normal – Množenje matrica jik





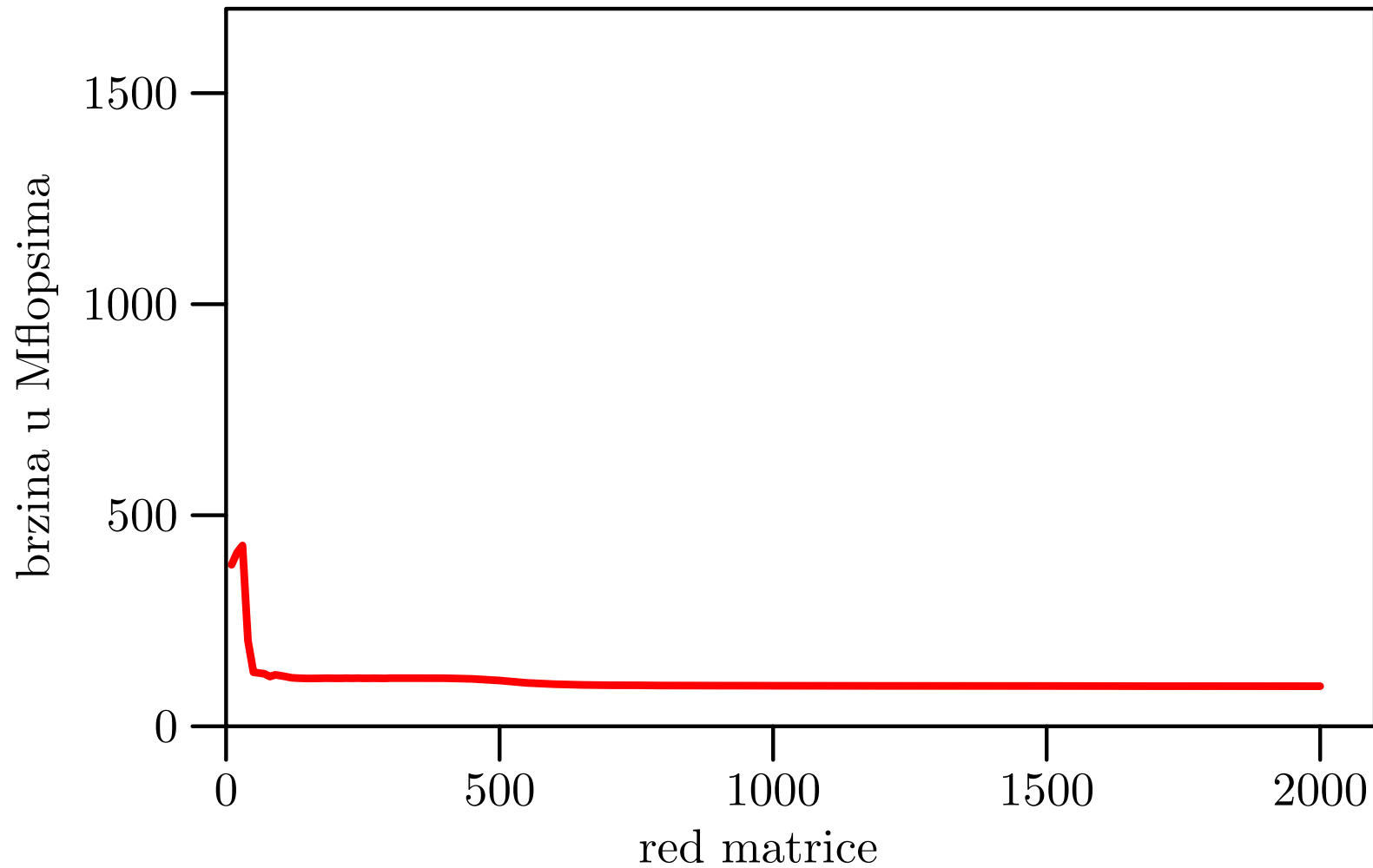
# BabyBlue, CVF, normal — jki

Pentium 4/660, 3.6 GHz, CVF, normal – Množenje matrica jki



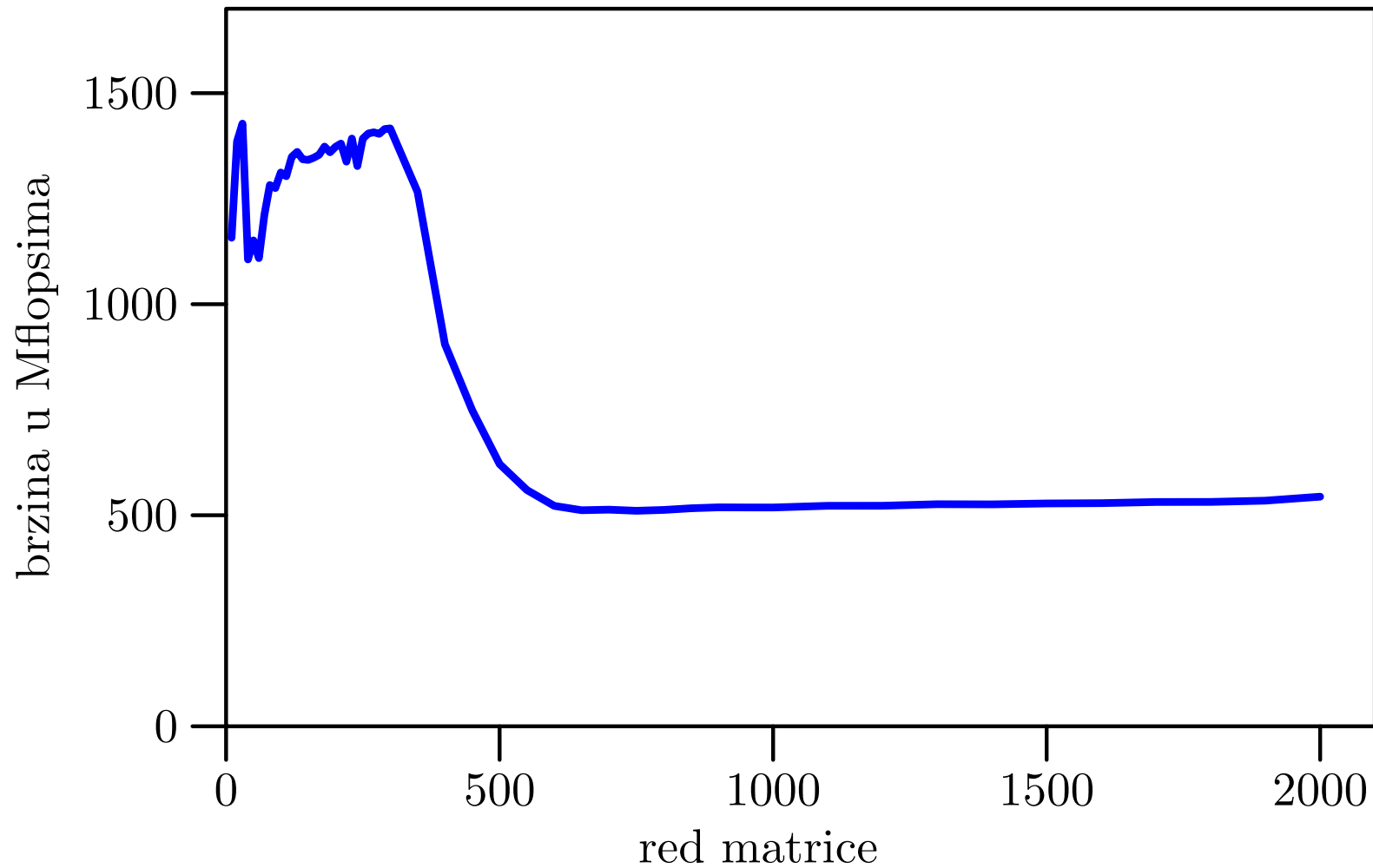
# BabyBlue, CVF, normal — *kij*

Pentium 4/660, 3.6 GHz, CVF, normal – Množenje matrica *kij*



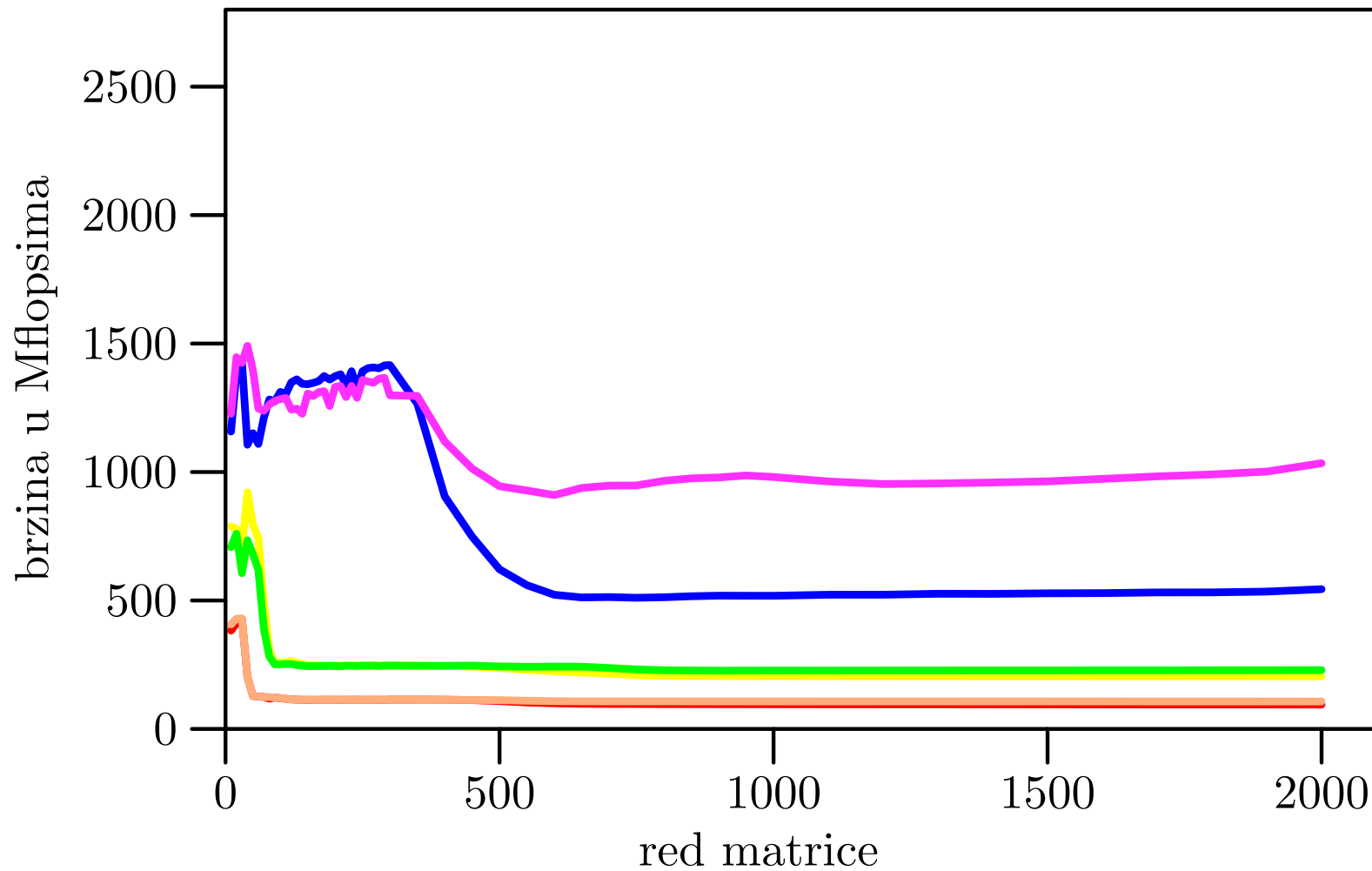
# BabyBlue, CVF, normal — *kji*

Pentium 4/660, 3.6 GHz, CVF, normal – Množenje matrica *kji*



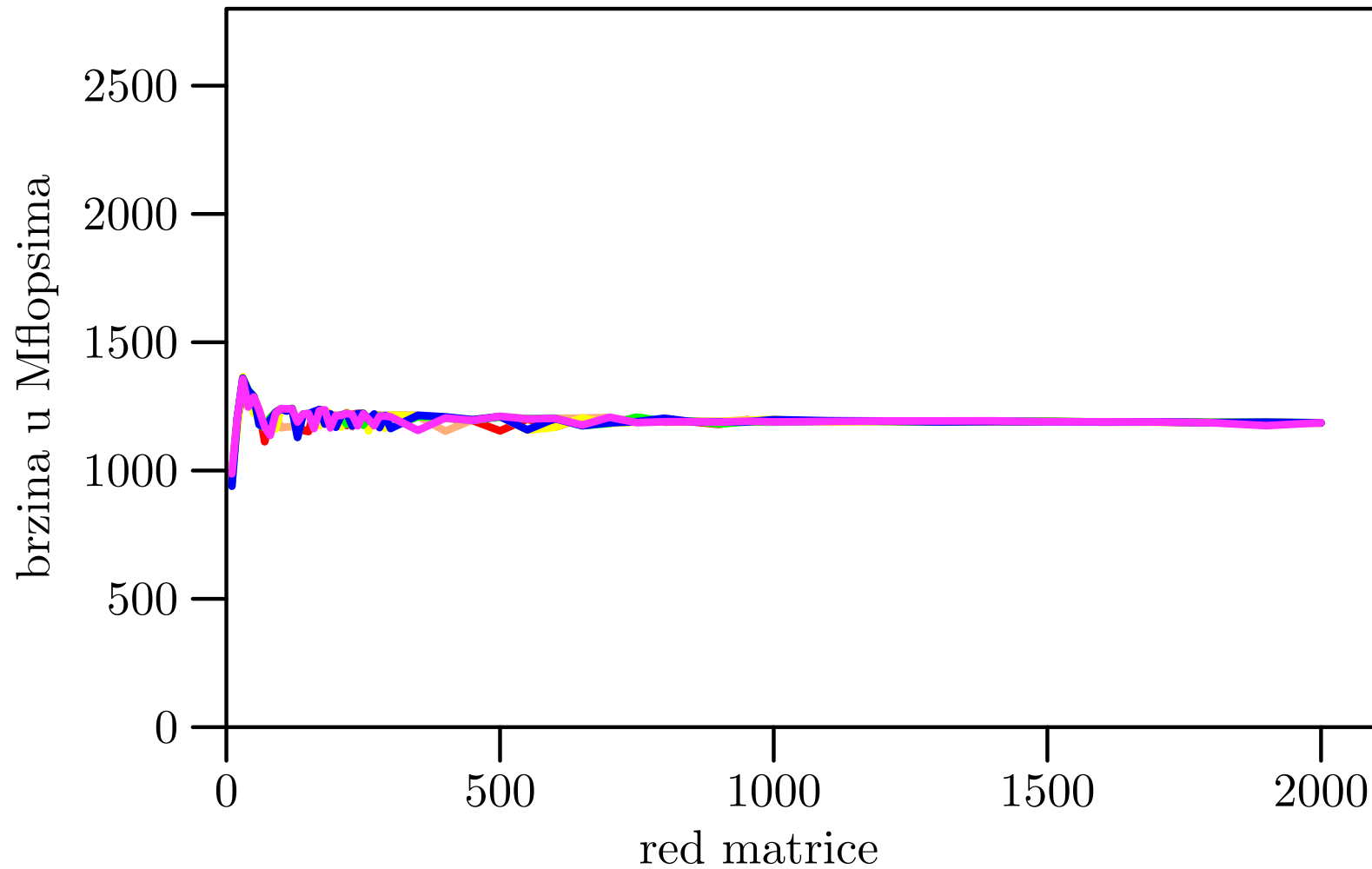
# BabyBlue, CVF, normal

Pentium 4/660, 3.6 GHz, CVF, normal – Množenje matrica



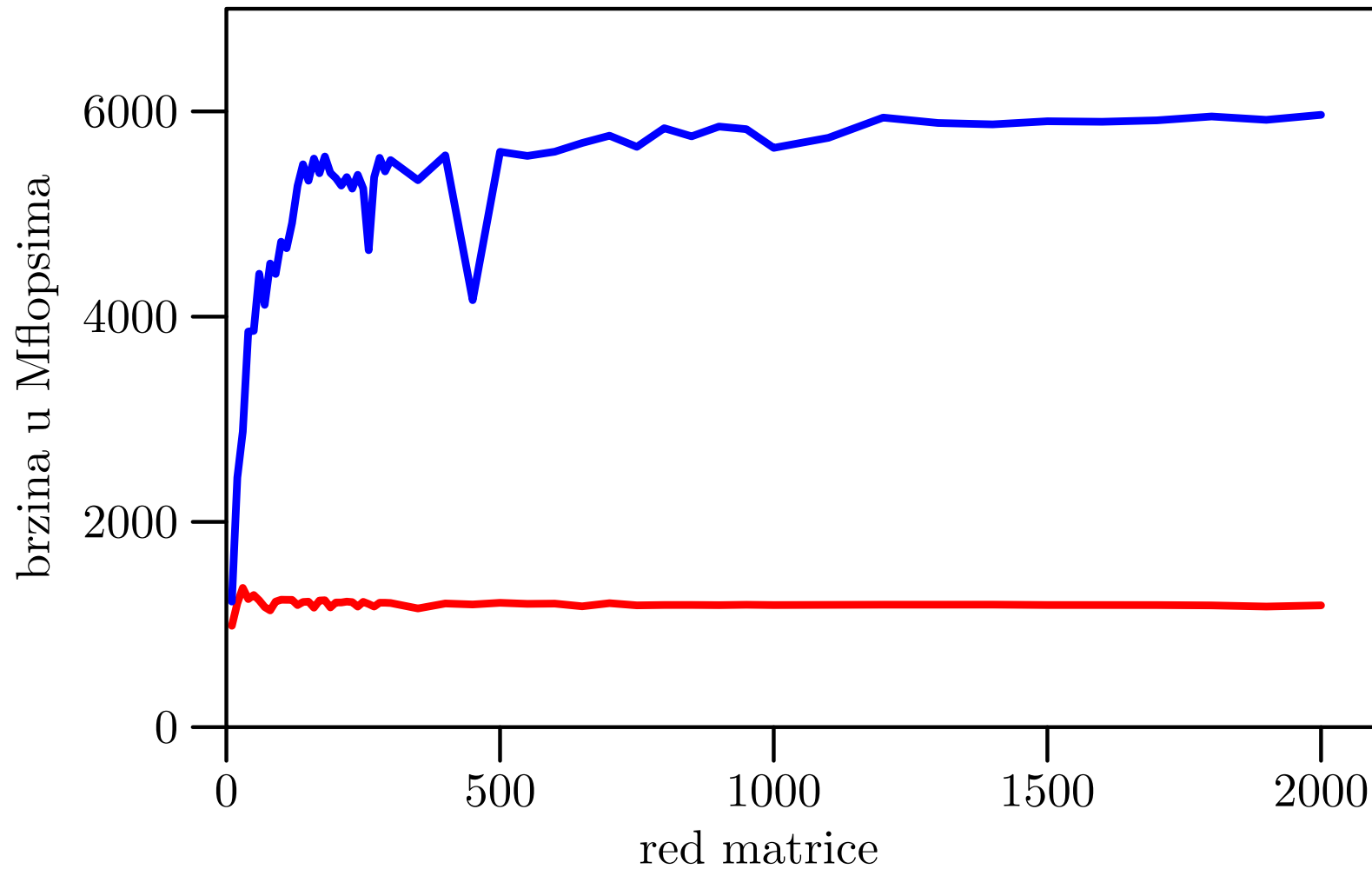
# BabyBlue, CVF, fast

Pentium 4/660, 3.6 GHz, CVF, fast – Množenje matrica



# BabyBlue, CVF, fast — najbrži i MKL

Pentium 4/660, 3.6 GHz, CVF, MKL – Množenje matrica



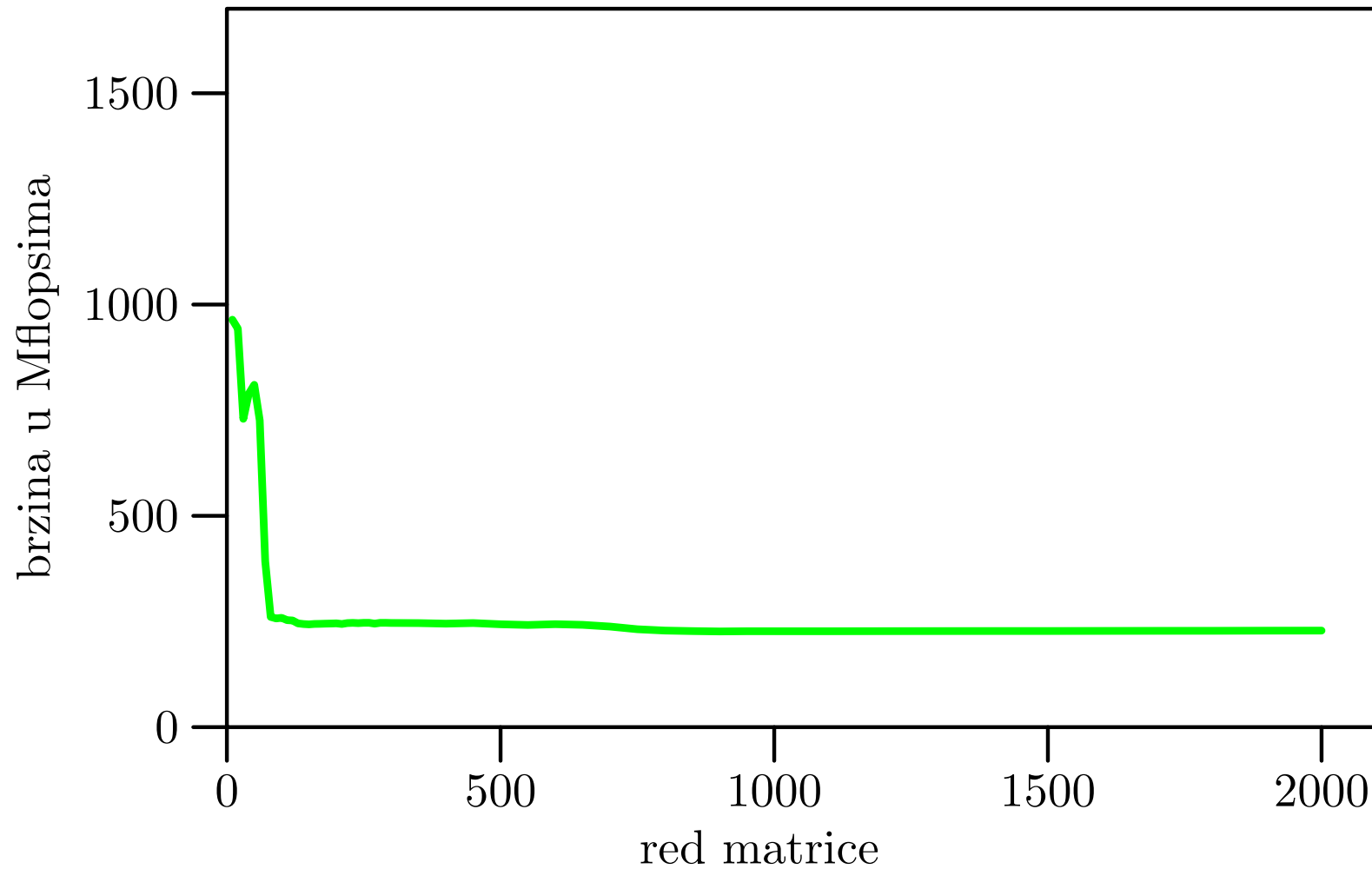
# BabyBlue, IVF, normal

## Intel Visual Fortran:

- normalna optimizacija:
  - prvo 6 pojedinačnih slika, leksikografskim redom, po petljama,
  - a zatim, zajednički graf za svih 6 petlji.
- fast optimizacija:
  - permutira petlje,
  - tako da imamo 3 para petlji s gotovo istom brzinom.
- Usporedba:
  - najbrže petlje jki u fast optimizaciji i
  - MKL-ovog algoritma DGEMM.

# BabyBlue, IVF, normal — ijk

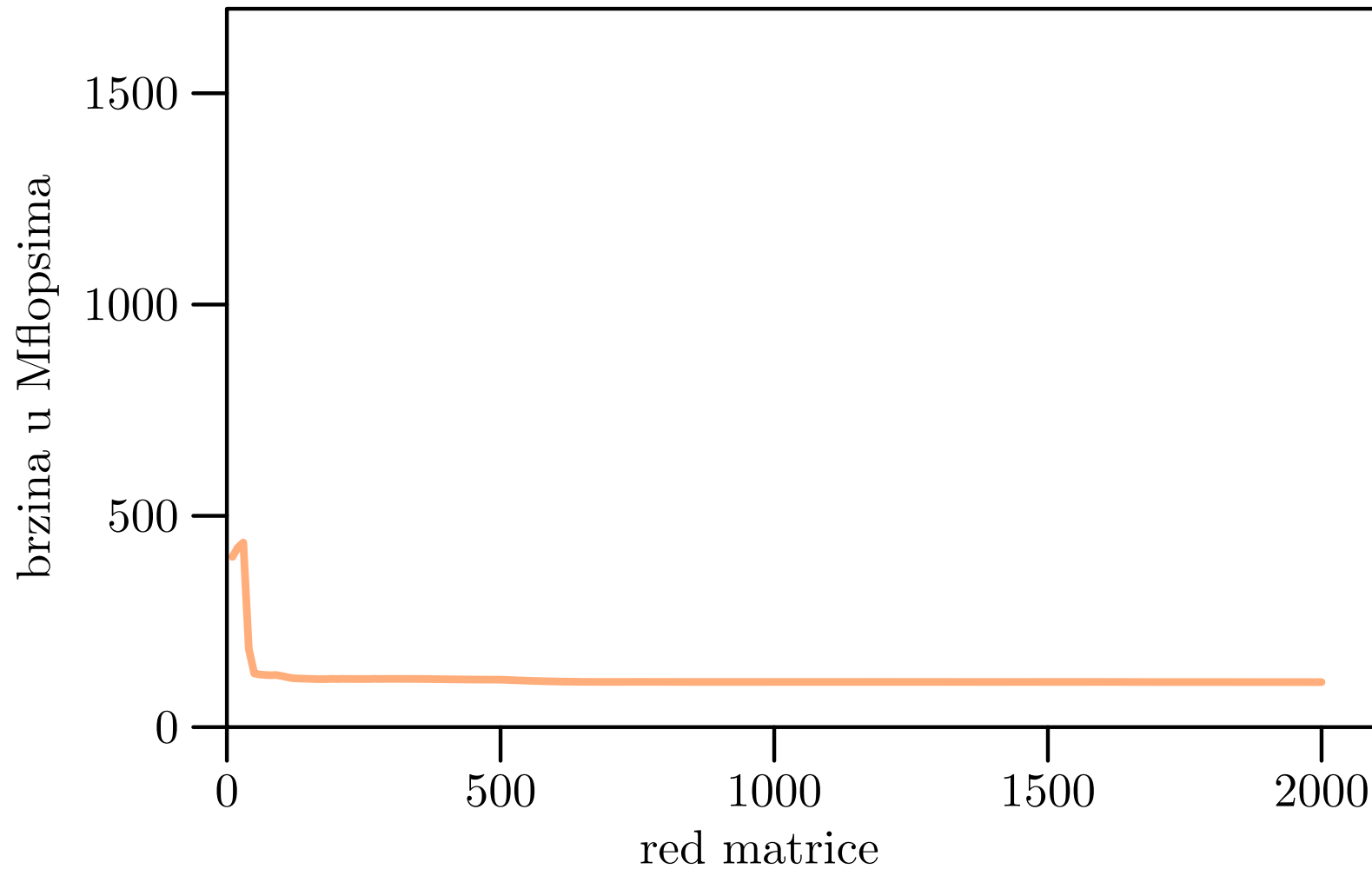
Pentium 4/660, 3.6 GHz, IVF, normal – Množenje matrica ijk





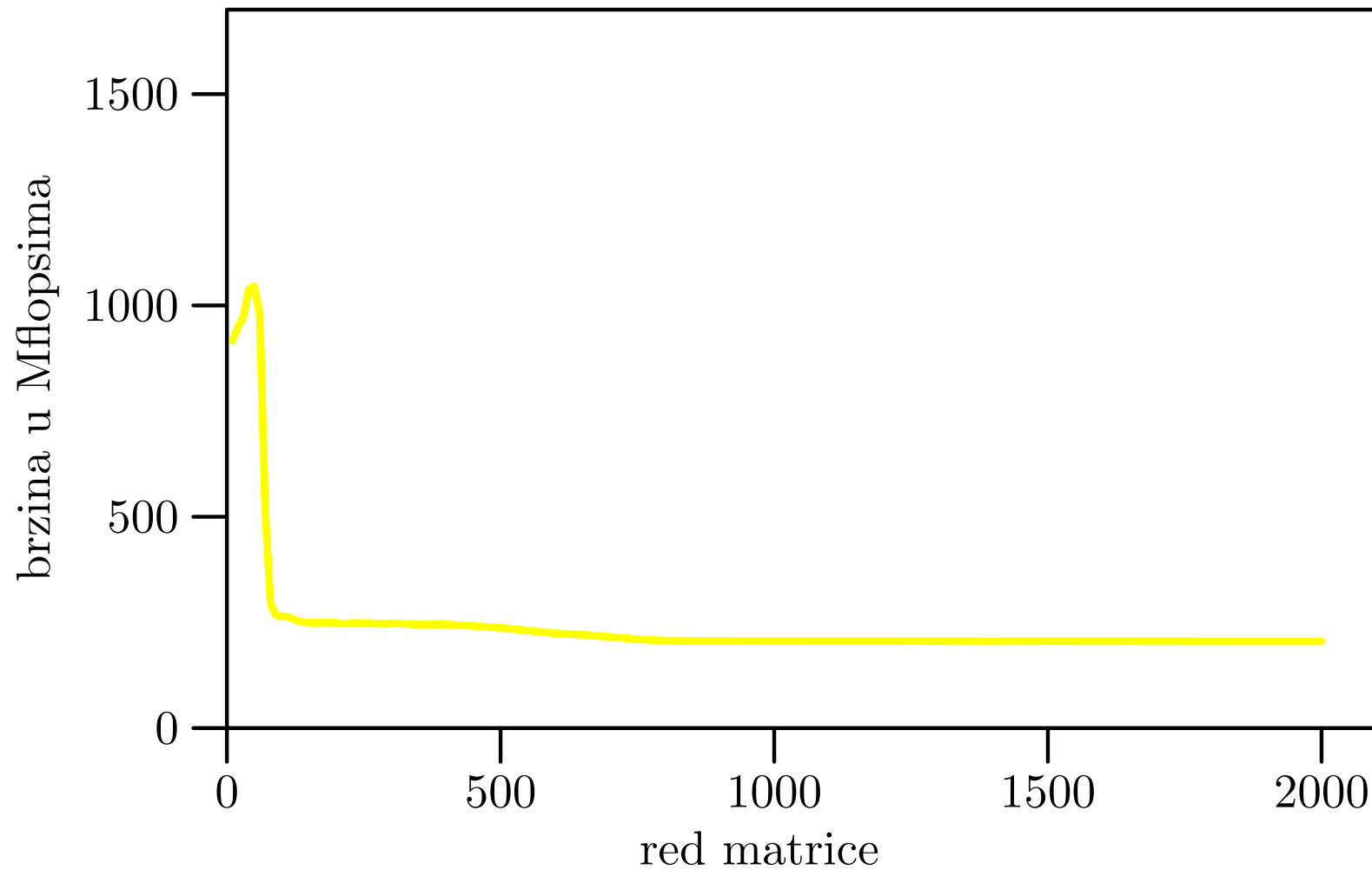
# BabyBlue, IVF, normal — ikj

Pentium 4/660, 3.6 GHz, IVF, normal – Množenje matrica ikj



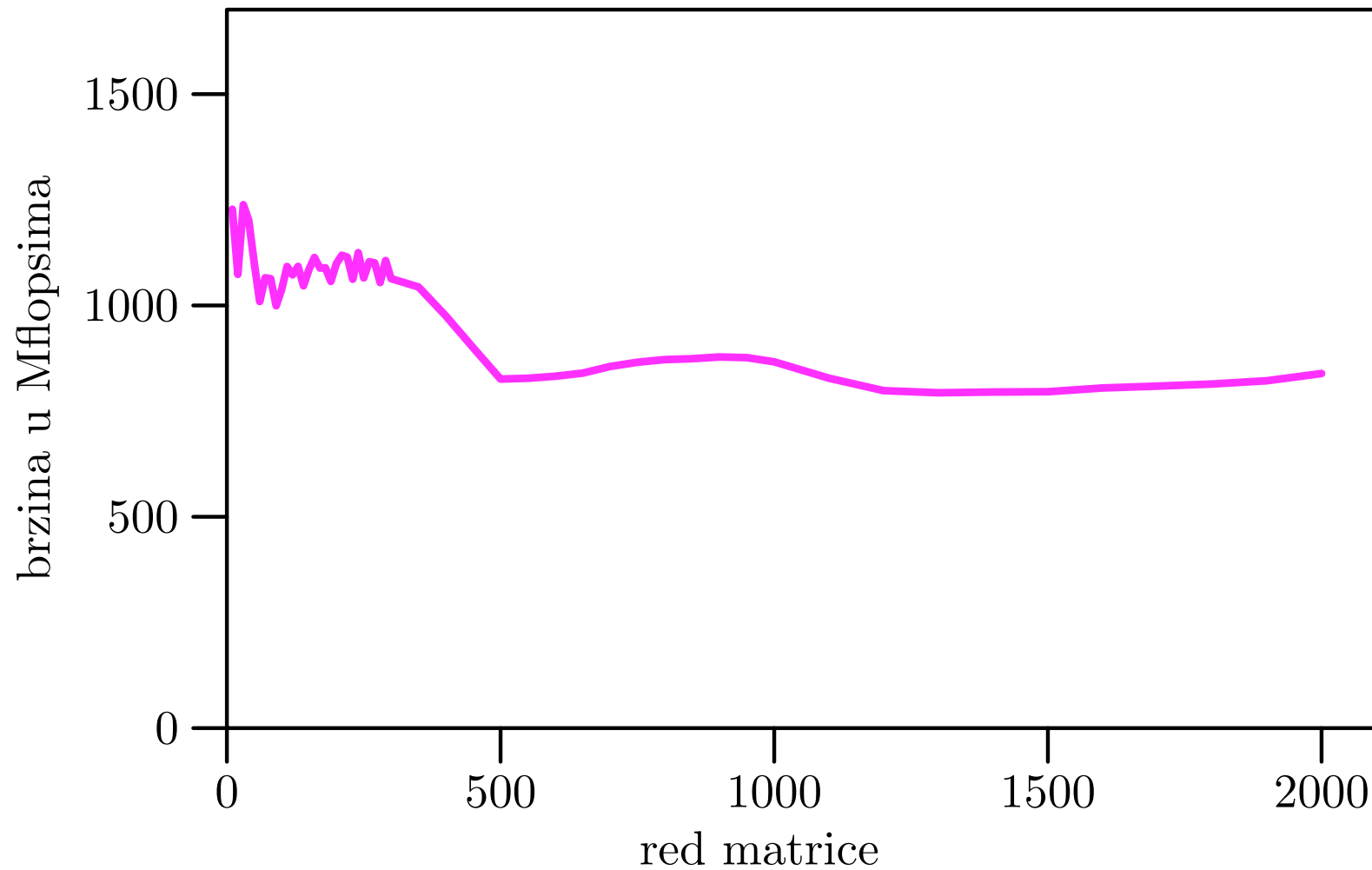
# BabyBlue, IVF, normal — jik

Pentium 4/660, 3.6 GHz, IVF, normal – Množenje matrica jik



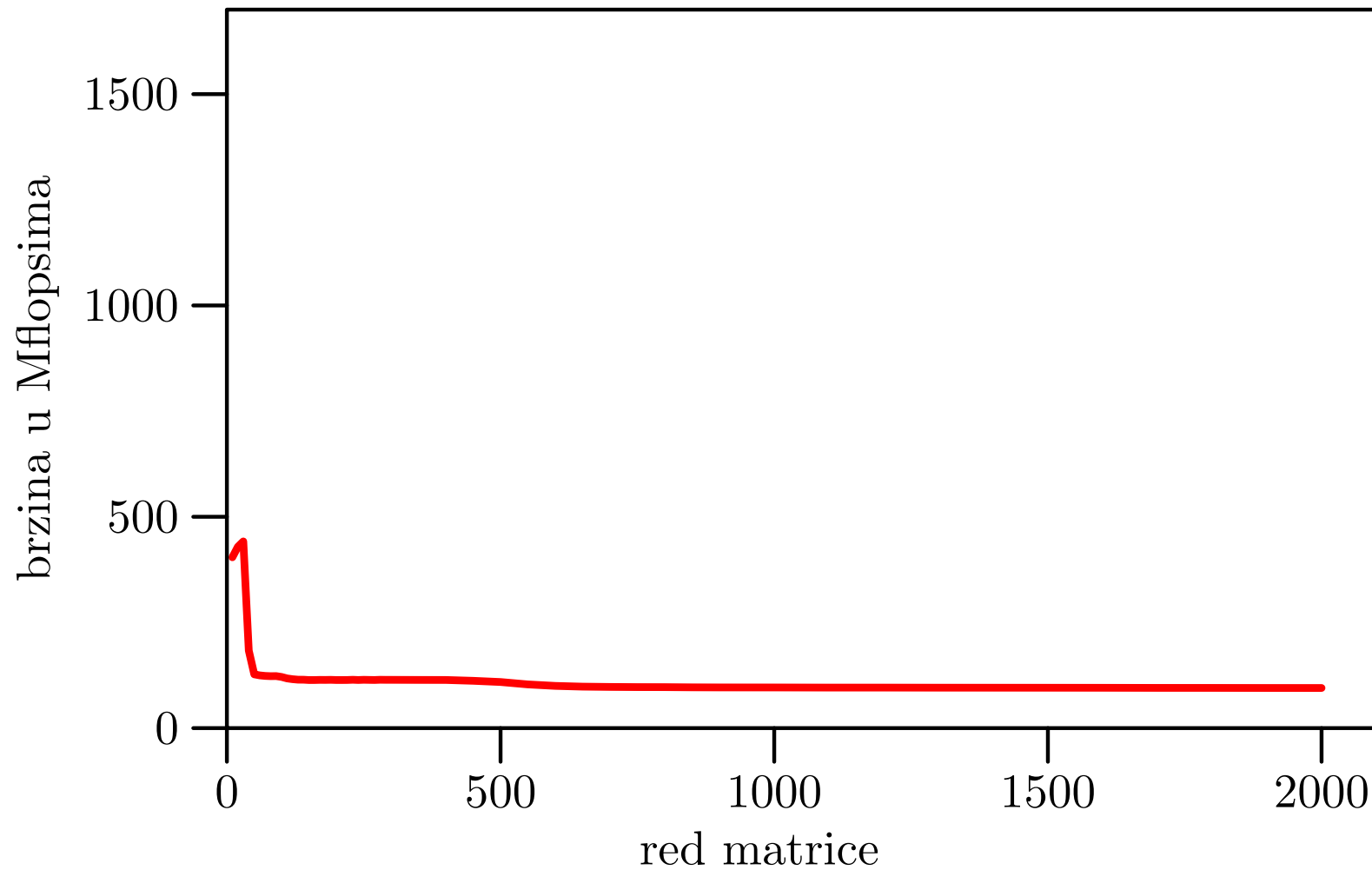
# BabyBlue, IVF, normal — jki

Pentium 4/660, 3.6 GHz, IVF, normal – Množenje matrica jki



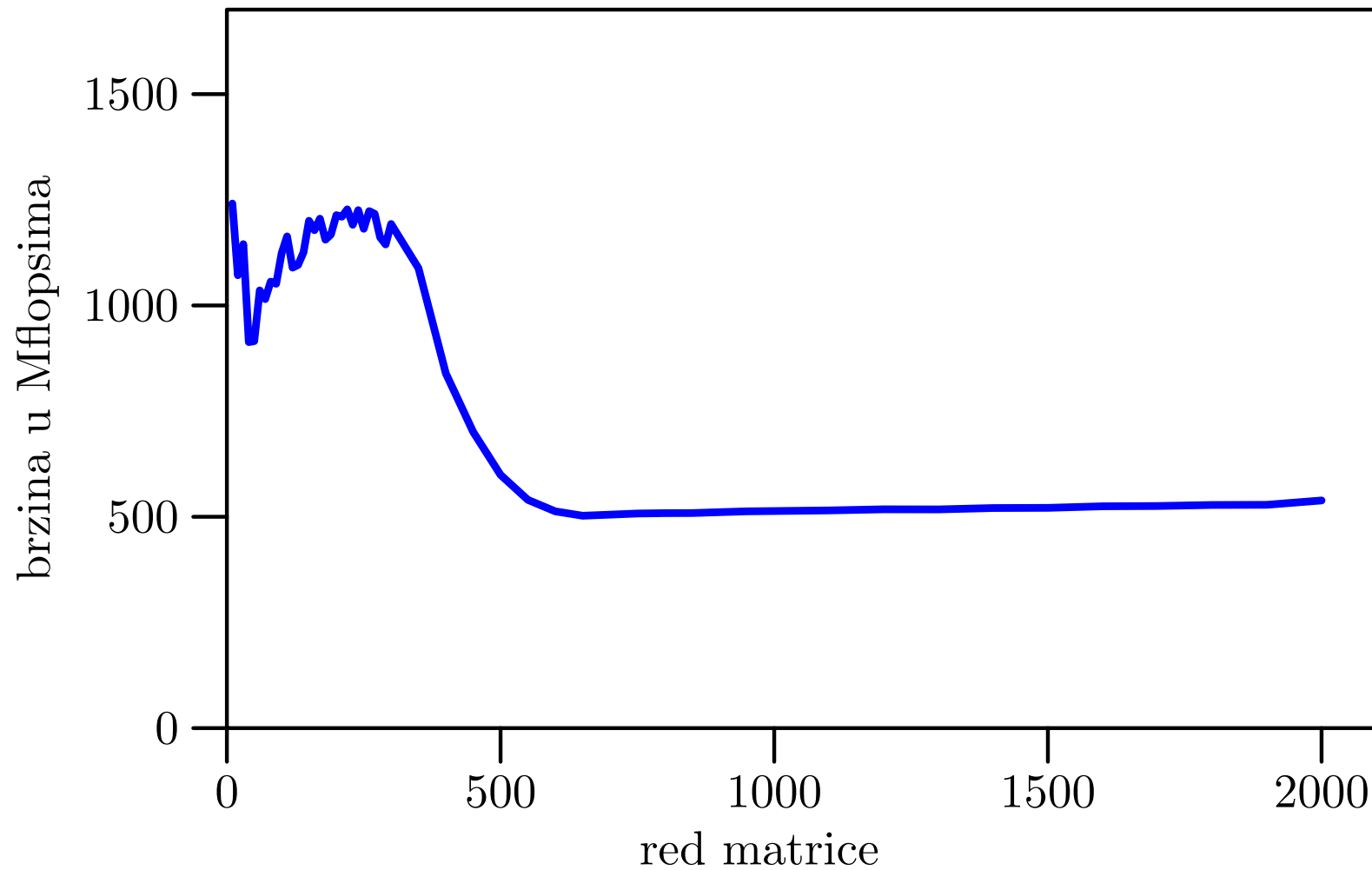
# BabyBlue, IVF, normal — kij

Pentium 4/660, 3.6 GHz, IVF, normal – Množenje matrica kij



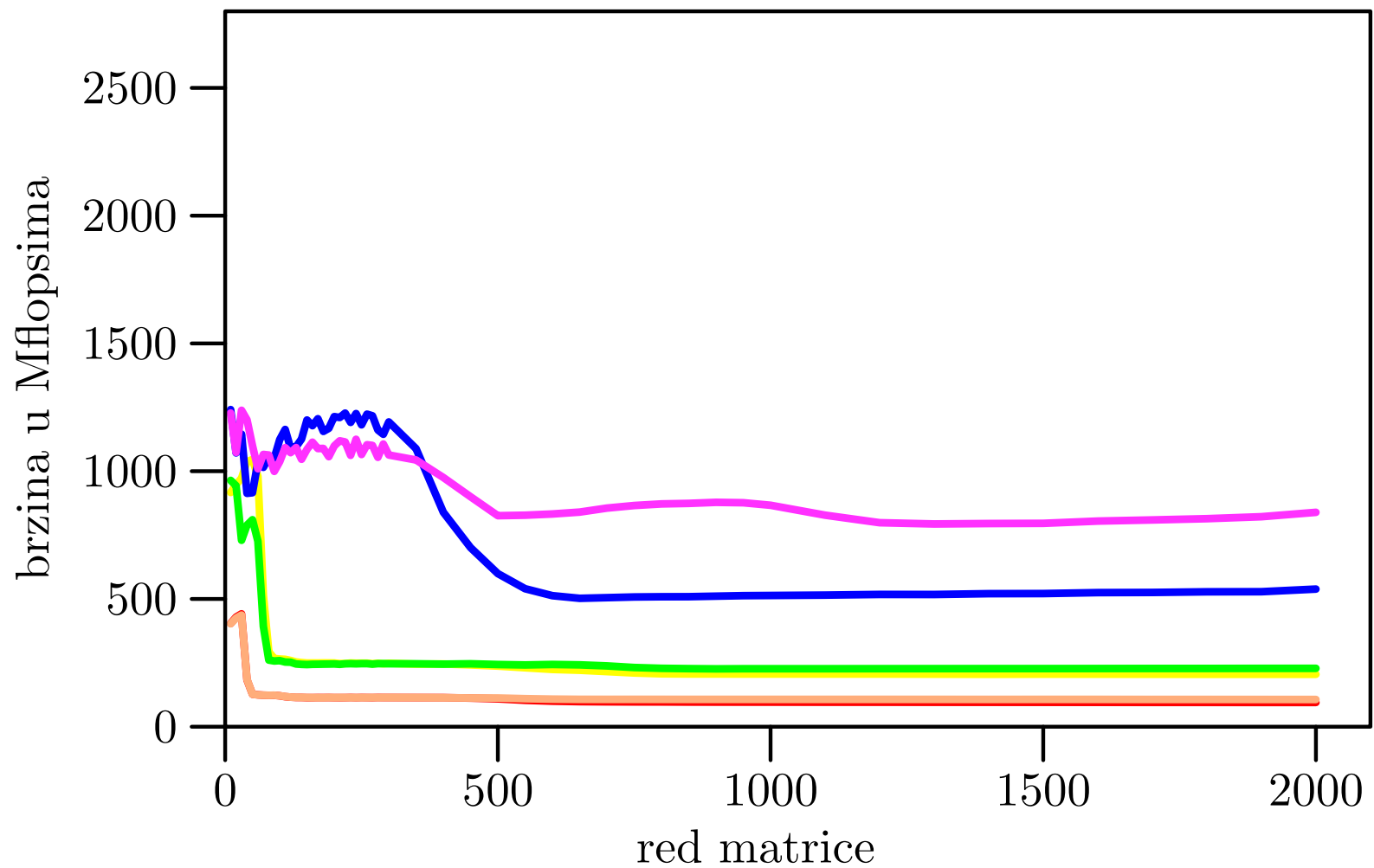
# BabyBlue, IVF, normal — kji

Pentium 4/660, 3.6 GHz, IVF, normal – Množenje matrica kji



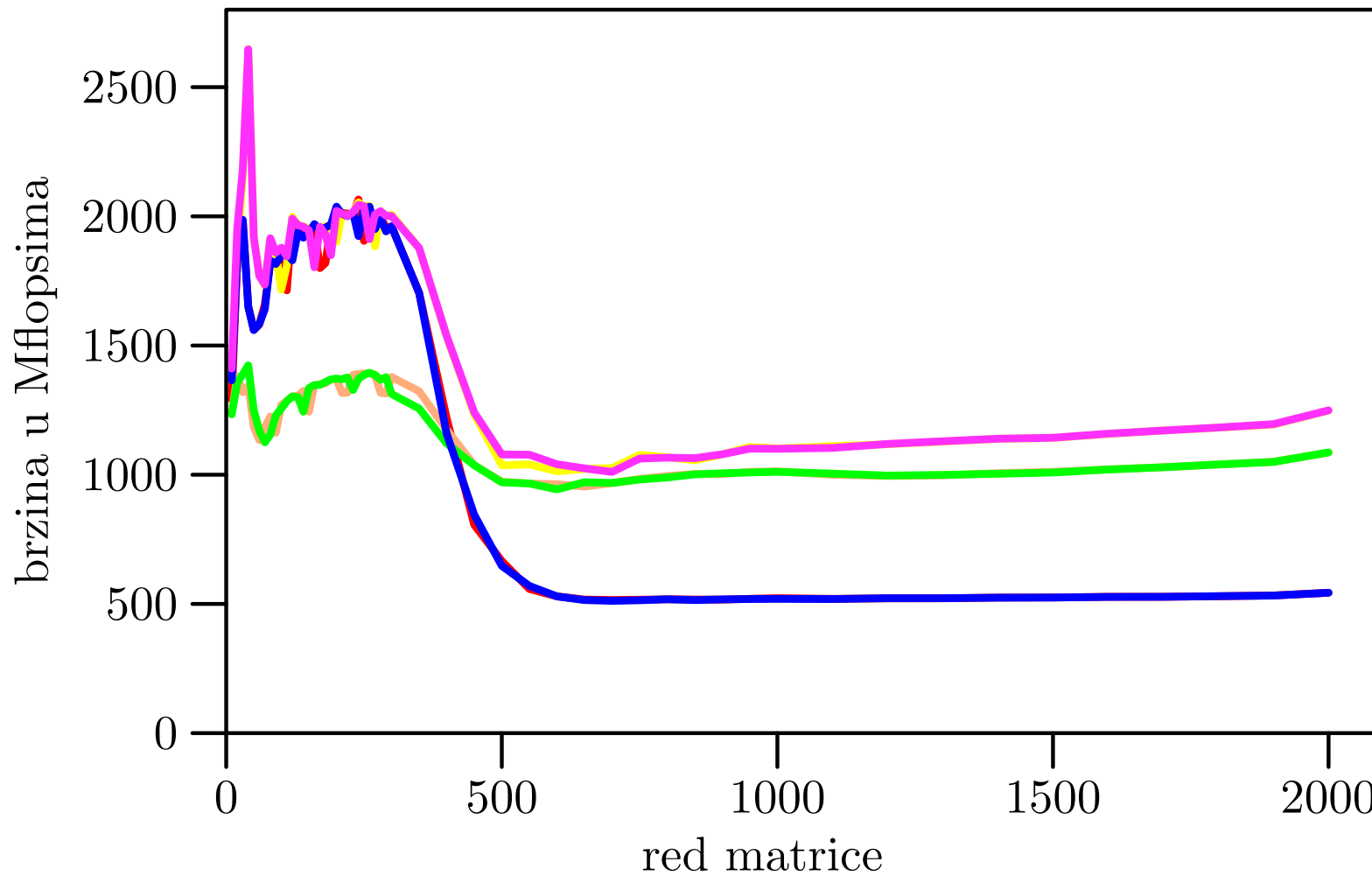
# BabyBlue, IVF, normal

Pentium 4/660, 3.6 GHz, IVF, normal – Množenje matrica



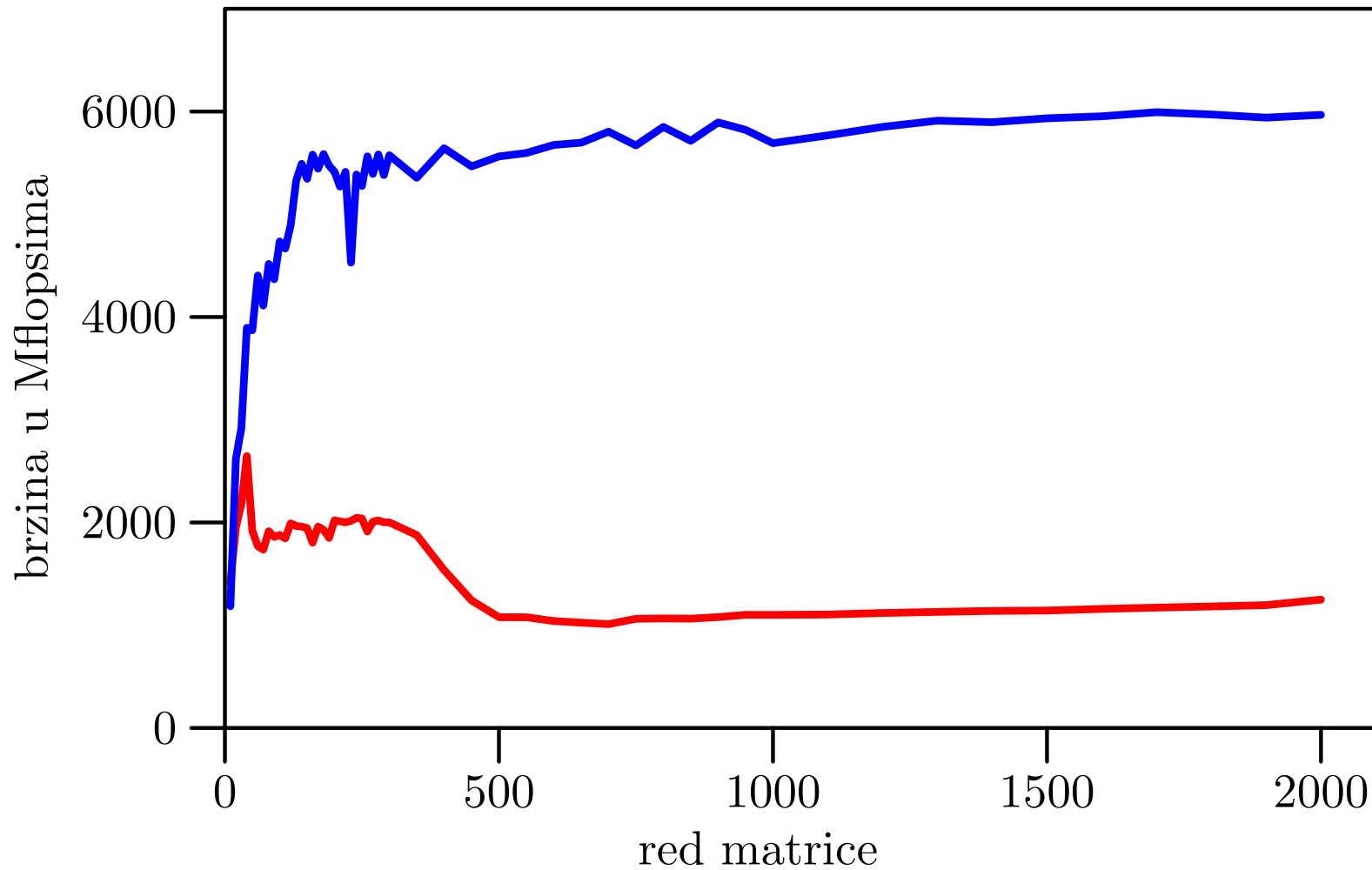
# BabyBlue, IVF, fast

Pentium 4/660, 3.6 GHz, IVF, fast – Množenje matrica



# BabyBlue, IVF, fast — najbrži i MKL

Pentium 4/660, 3.6 GHz, IVF, MKL – Množenje matrica





## Tablica brzina za velike $n$

Usporedba brzina (u Mflops) samo na BabyBlue:

- po petljama (uključivo i MKL),
- za normal i fast opcije kod oba kompilera.

Petlja	normal CVF	normal IVF	fast CVF	fast IVF
ijk	229.0	228.4	1186.3	1086.4
ikj	106.7	106.5	1186.5	1086.0
jik	204.8	205.1	1185.3	1248.7
jki	1034.2	839.0	1186.4	1249.1
kij	95.0	94.9	1185.1	543.3
kji	544.0	538.6	1185.7	543.3
MKL	5945.7	5967.3	5966.5	5967.6

## Ostala računala

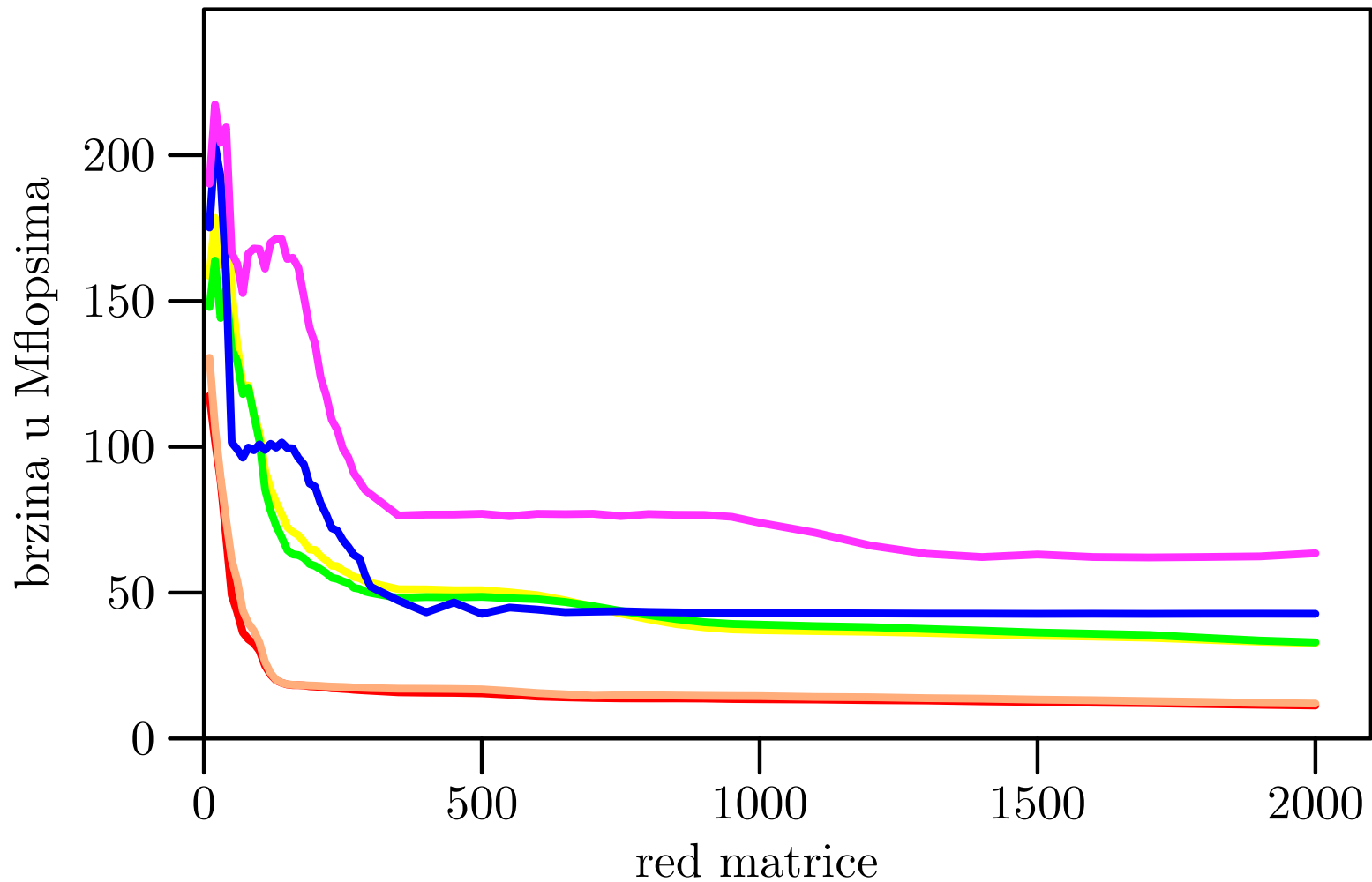
Vrlo **slično** ponašanje **brzina** za **petlje** vidi se i na ostalim računalima.

Grafovi su “**skraćeni**” tako da sadrže redom:

- usporedbu **brzina** svih **6** petlji za **normal** i **fast** opcije kompilera (samo CVPF),
- usporedbu **najbrže fast** petlje **MKL-a**.

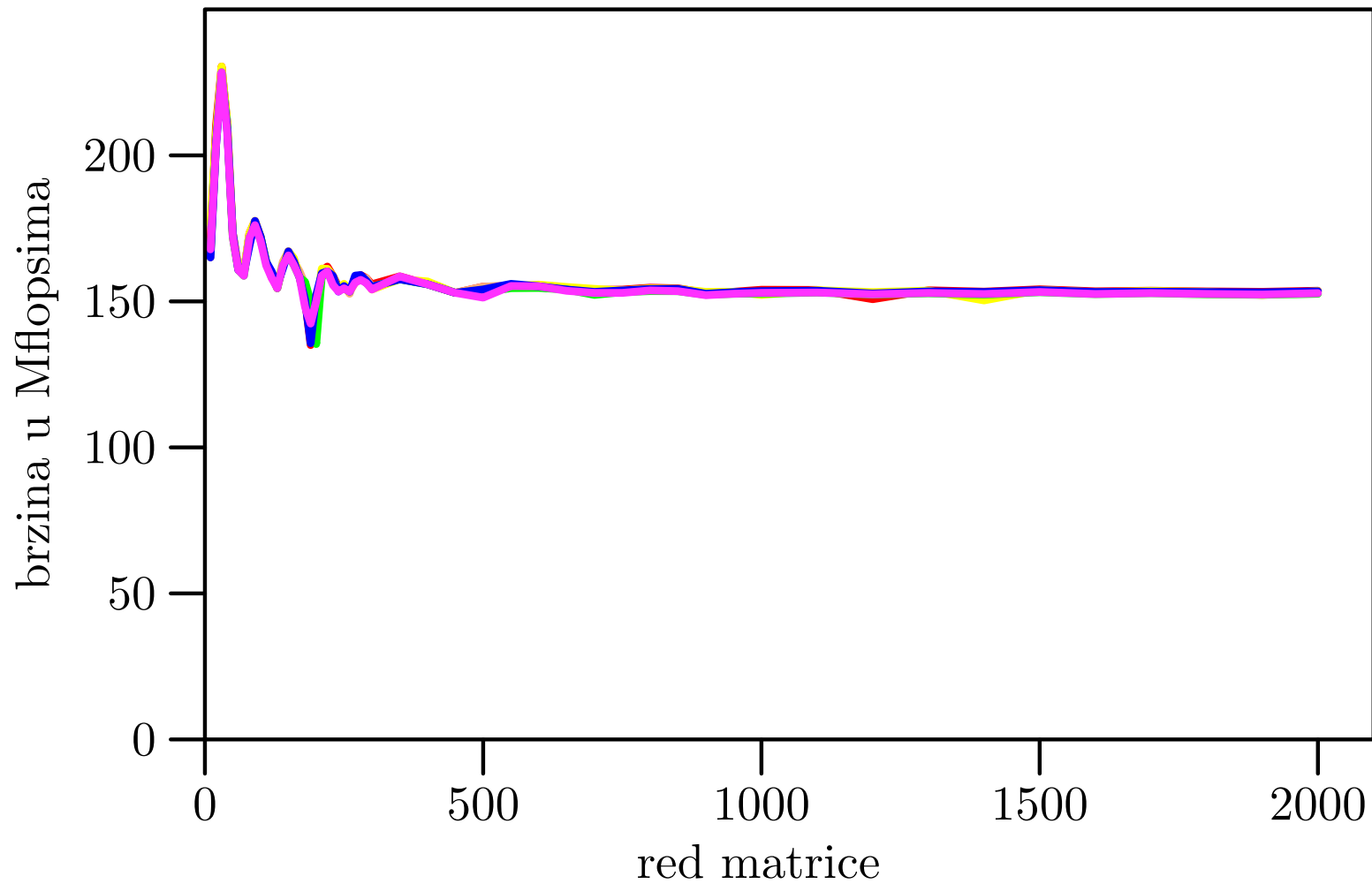
# Klamath5, CVF, normal

Pentium III, 500 MHz, CVF, normal – Množenje matrica



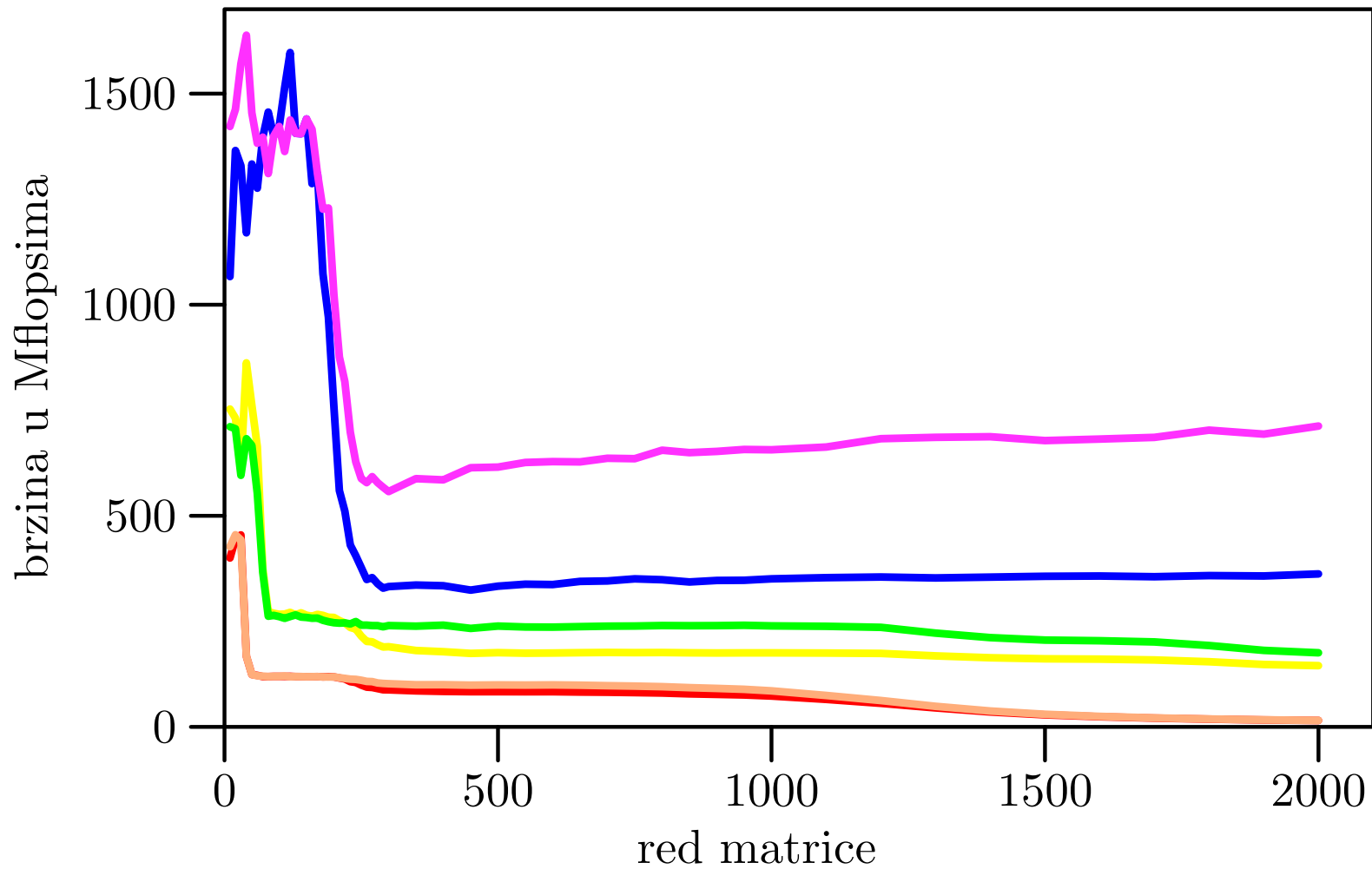
# Klamath5, CVF, fast

Pentium III, 500 MHz, CVF, fast – Množenje matrica



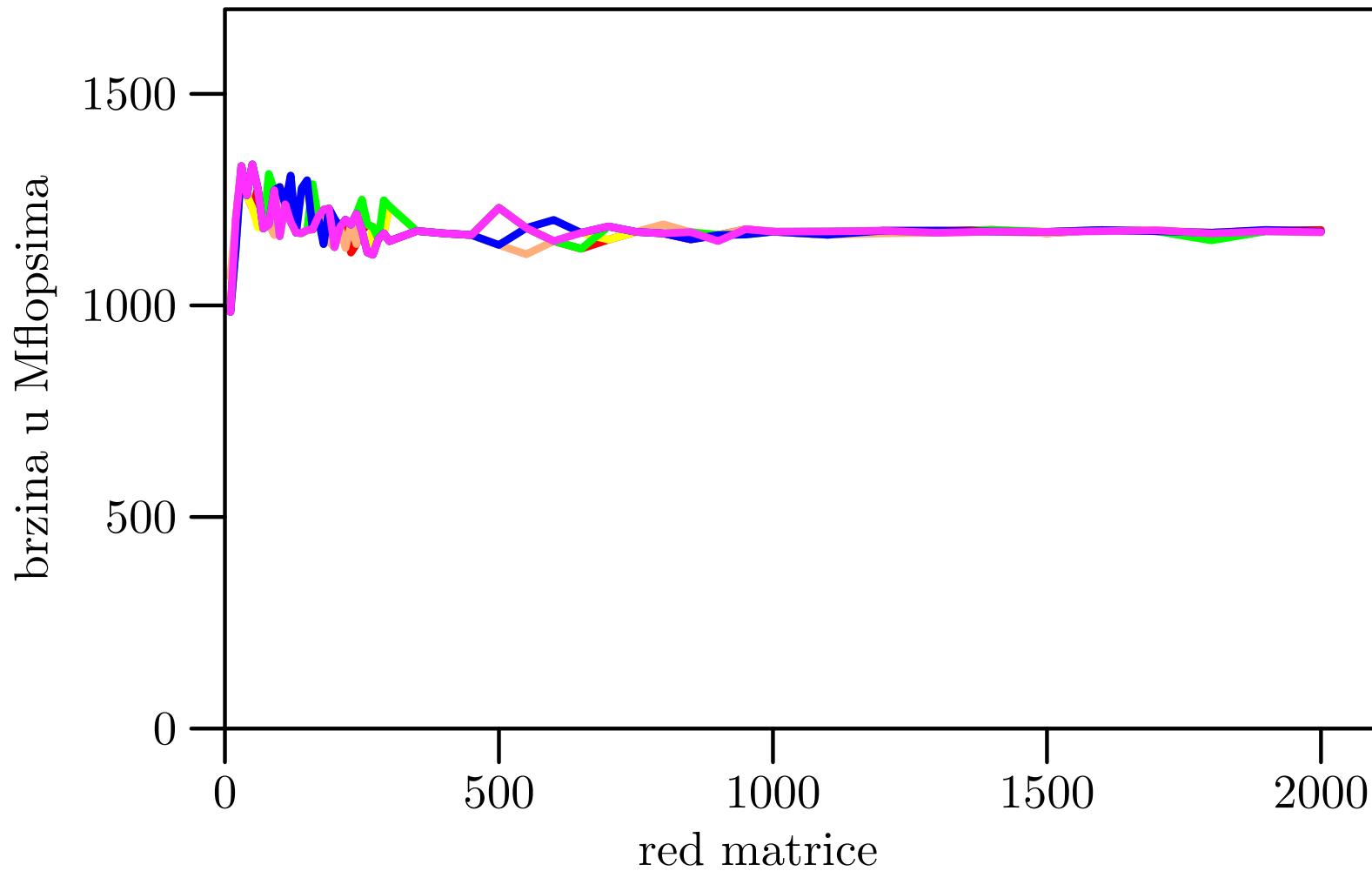
# Veliki, CVF, normal

Pentium 4, 3.0 GHz, CVF, normal – Množenje matrica



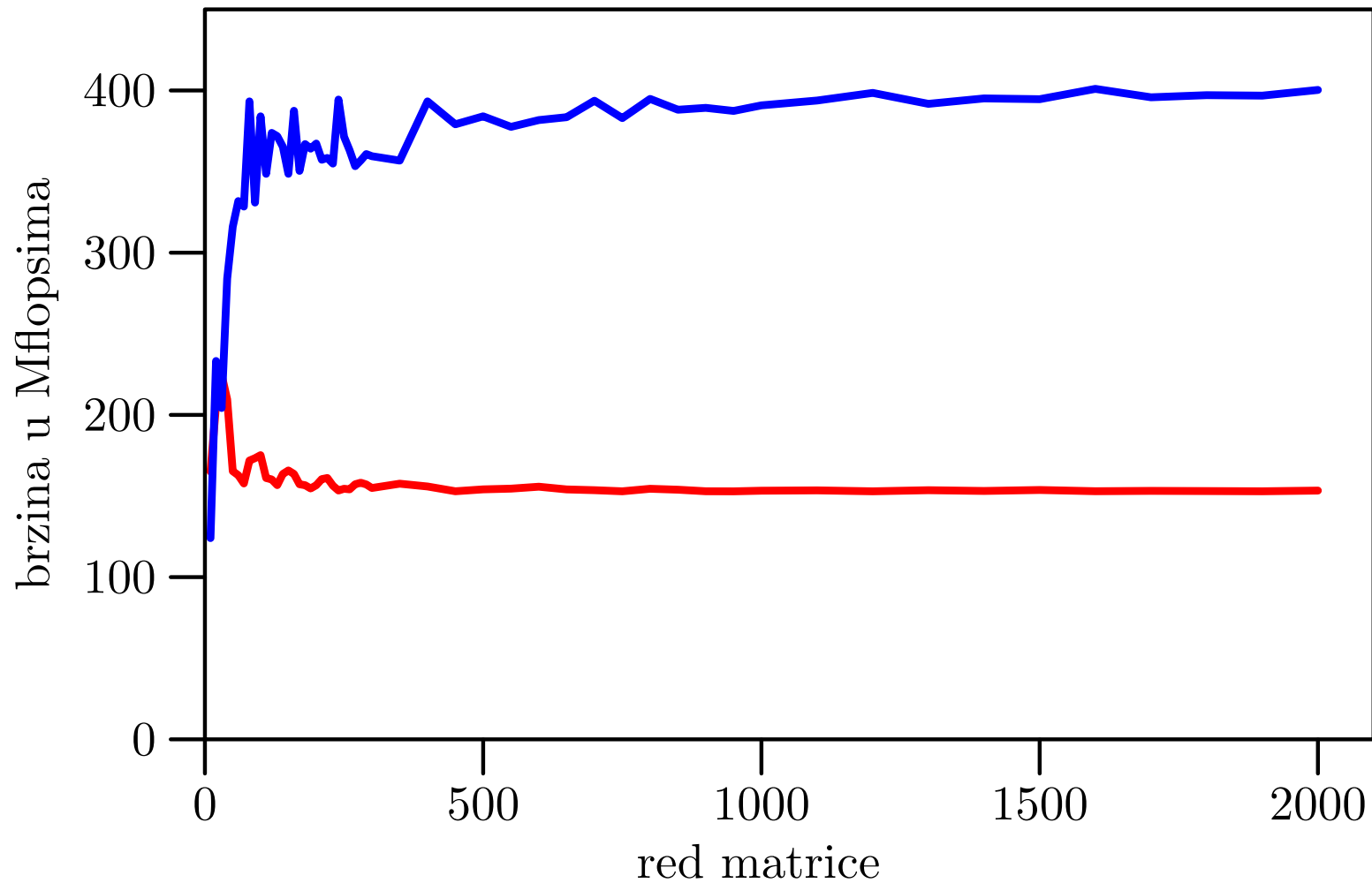
# Veliki, CVF, fast

Pentium 4, 3.0 GHz, CVF, fast – Množenje matrica



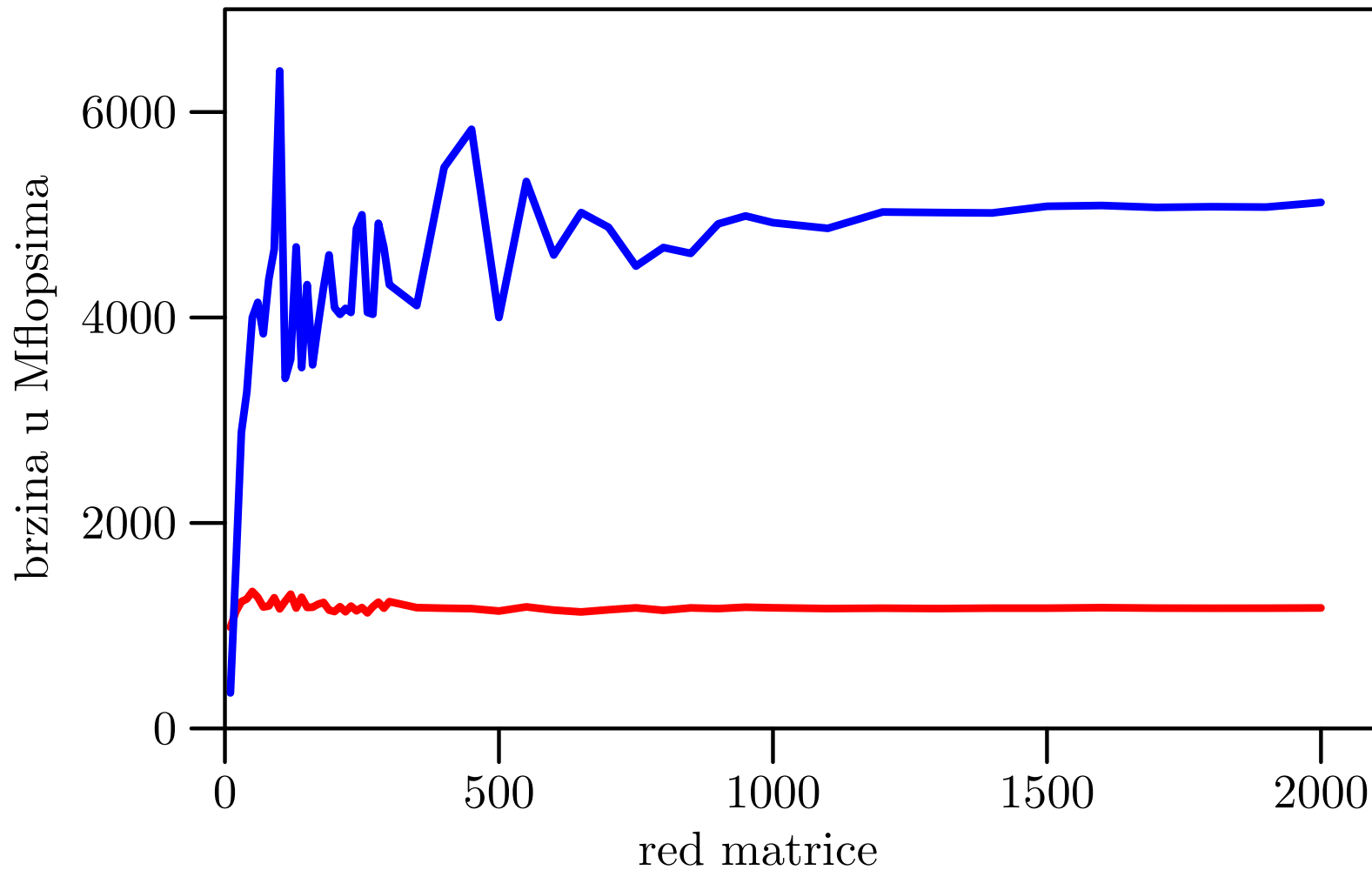
# Klamath5, CVF, fast — najbrži i MKL

Pentium III, 500 MHz, CVF, MKL – Množenje matrica



# Veliki, CVF, fast — najbrži i MKL

Pentium 4, 3.0 GHz, CVF, MKL – Množenje matrica





## Komentar rezultata

Kod množenja matrica, za razliku od zbrajanja,

- svaki ulazni podatak koristimo puno puta, (preciznije, točno  $n$  puta).

Zato brzina cache memorije može doći do izražaja, pa možemo dobiti

- bitno veće brzine nego kod zbrajanja.

Cache memorija je “glavni krivac” za:

- razlike u brzinama između raznih varijanti, i
- povećanu brzinu za male  $n$ -ove.

Ponavljanje eksperimenta ima neku ulogu samo za vrlo male redove  $n$ . Osim toga, za  $n \geq 450$  nema ponavljanja.

## Komentar rezultata (nastavak)

Brže su one varijante koje

- učestalije koriste iste podatke, dok su oni još u cacheu.

Dokaz: Cache se “puni” u “blokovima”, kako su matrice spremljene. Najbrža bi trebala biti ona varijanta koja

- sekvencijalno prolazi kroz elemente u sve 3 matrice u “unutarnjoj” naredbi

---

$$c(i, j) = c(i, j) + a(i, k) * b(k, j)$$

---

U Fortranu, zbog spremanja matrice po stupcima, prvi indeks se brže mijenja. Zato mora biti:

- i unutar j, i unutar k, k unutar j.

Dakle, najbrža varijanta algoritma je jki, što zaista i je!

## Komentar rezultata (nastavak)

Zadnji argument da je “krivac” cache memorija.

Konstruktivni dokaz: “Blokovskom” realizacijom algoritma

- za *velike*  $n$  možemo postići *gotovo iste* brzine kao i za *male*  $n$  (tj. spriječiti pad brzine).

Ovo, naravno, ide samo onda kad

- za *velike*  $n$  dobijemo *pad brzine*.

U protivnom, compiler se “već pobrinuo” da optimalno iskoristi cache.

Primjer za IVF da to *radi* za *normal*, pa čak i za *fast* opciju.

# Blokovsko množenje matrica primjer

## Blokovsko množenje matrica — primjer

IVF s *normal* opcijom za *jik* petlju daje brzine:

- 1050 MFlops za  $n \leq 50$ ,
- 205 MFlops za velike  $n$ .

IVF s *normal* opcijom za *jki* petlju daje brzine:

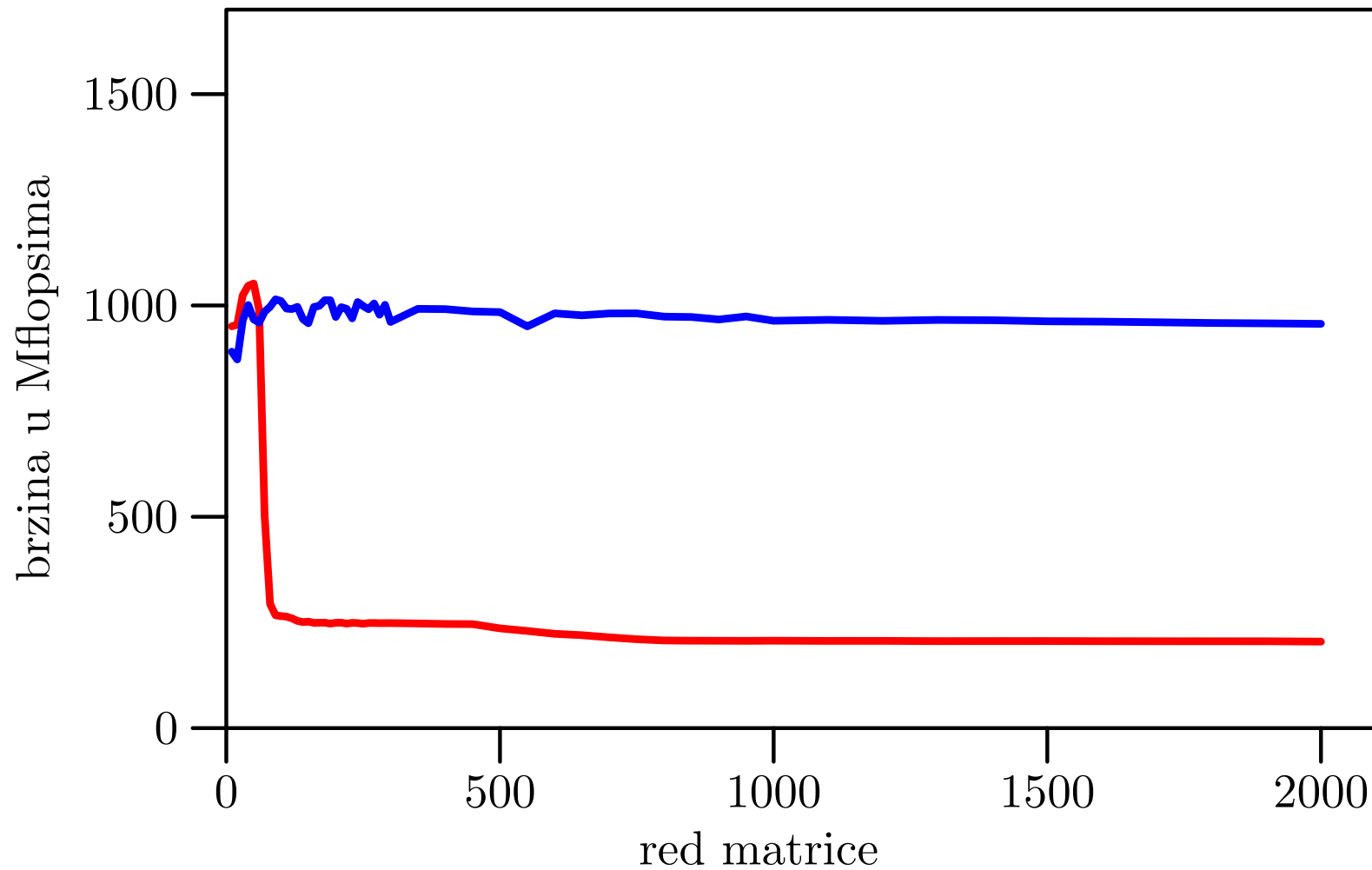
- 1100 MFlops za  $n \leq 300$ ,
- 840 MFlops za velike  $n$ .

IVF s *fast* opcijom za *jki* petlju daje brzine:

- 2000 MFlops za  $n \leq 300$ ,
- 1250 MFlops za velike  $n$ .

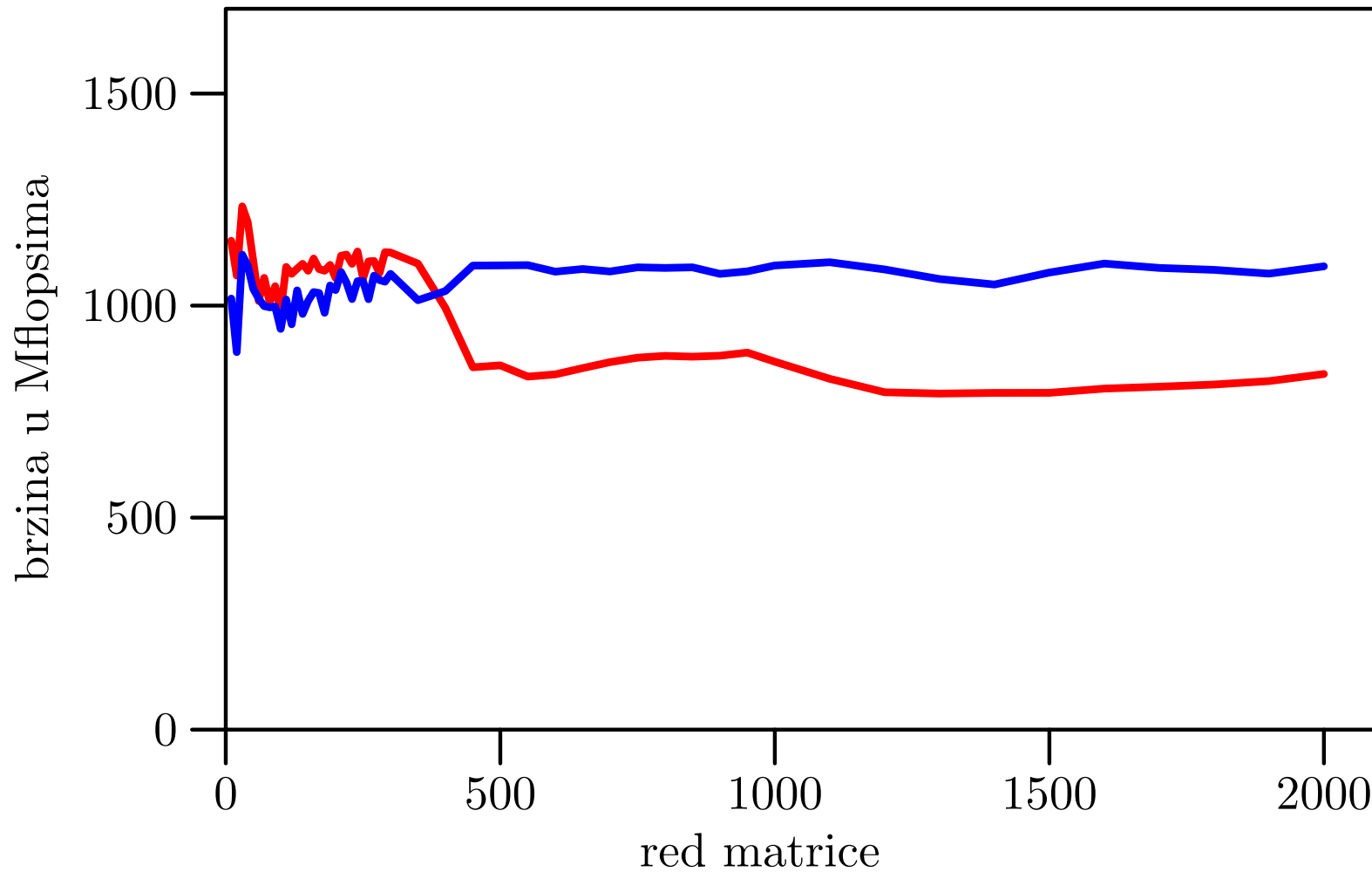
# BabyBlue, IVF, normal — jik *obični i blok (50)*

Pentium 4/660, 3.6 GHz, IVF, normal – Množ. mat. jik (50)



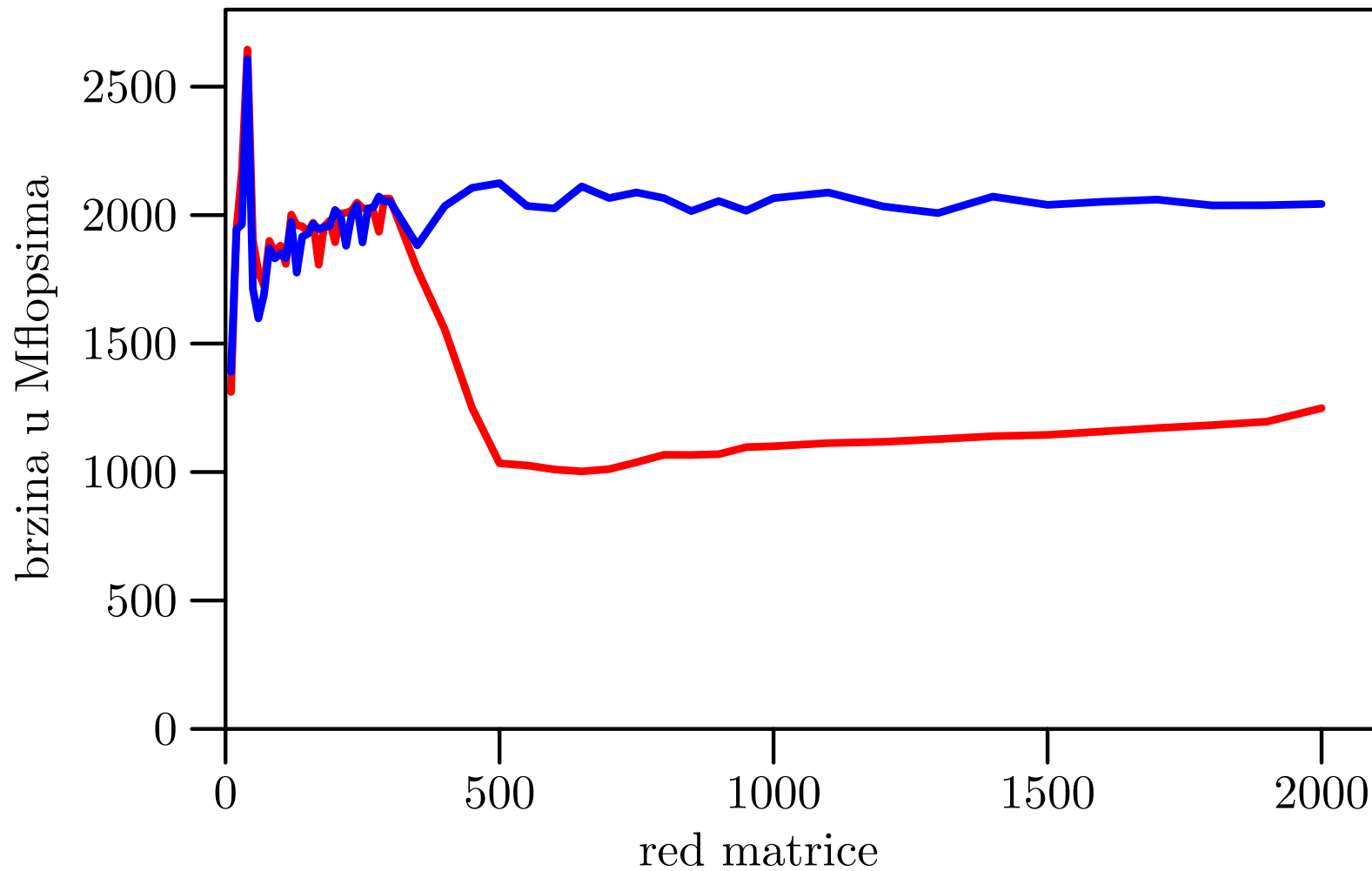
# BabyBlue, IVF, normal — jki *obični i blok (300)*

Pentium 4/660, 3.6 GHz, IVF, normal – Množ. mat. jki (300)



# BabyBlue, IVF, fast — jki *obični* i blok (300)

Pentium 4/660, 3.6 GHz, IVF, fast – Množ. mat. jki (300)





# Blokovsko množenje matrica

# Množenje matrica

**Problem:** Zadan je prirodni broj  $n \in \mathbb{N}$  i 3 matrice  $A$ ,  $B$  i  $C$ , reda  $n$ . Treba izračunati izraz

$$C := C + A * B.$$

Znamo da je realizacija po **elementima** trivijalna

$$c_{ij} := c_{ij} + \sum_{k=1}^n a_{ik} \cdot b_{kj},$$

za sve indekse

$$i = 1, \dots, n, \quad j = 1, \dots, n.$$

Dakle, “programski” — treba “zavrtiti” **tri** petlje.

# Množenje matrica — realizacija po elementima

Programska realizacija na “skalarnoj” razini (po elementima) ima ovaj opći oblik:

- 3 petlje po  $i$ ,  $j$ ,  $k$ , svaka od 1 do  $n$ ,
- operacija unutar tih petlji je

$$c_{ij} := c_{ij} + a_{ik} \cdot b_{kj},$$

tj. množenje i zbrajanje skalara.

Ove tri petlje smijemo permutirati pa dobivamo 6 različitih varijanti osnovnog algoritma:

- $ijk$ ,  $ikj$ ,  $jik$ ,  $jki$ ,  $kij$ ,  $kji$ .

# Množenje matrica — podjela na blokove

Matrice  $A$  i  $B$  možemo podijeliti na blokove

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1r} \\ A_{21} & A_{22} & \cdots & A_{2r} \\ \vdots & \vdots & \cdots & \vdots \\ A_{p1} & A_{p2} & \cdots & A_{pr} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} & \cdots & B_{1q} \\ B_{21} & B_{22} & \cdots & B_{2q} \\ \vdots & \vdots & \cdots & \vdots \\ B_{r1} & B_{r2} & \cdots & B_{rq} \end{bmatrix}.$$

Ako su blokovi  $A_{ik}$  i  $B_{kj}$  takvi da se mogu množiti za sve indekse  $i, j, k$ , onda operaciju  $C = C + A * B$  možemo izračunati “po blokovima”, gdje je

$$C_{ij} = C_{ij} + \sum_{k=1}^r A_{ik} * B_{kj}, \quad i = 1, \dots, p, \quad j = 1, \dots, q.$$

# Množenje matrica — blokovi (nastavak)

Podjela matrica  $A$  i  $B$  na blokove koji se mogu množiti inducira podijelu matrice  $C$  na blokove

$$C = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1q} \\ C_{21} & C_{22} & \cdots & C_{2q} \\ \vdots & \vdots & \cdots & \vdots \\ C_{p1} & C_{p2} & \cdots & C_{pq} \end{bmatrix} .$$

Pojednostavljenje: sve tri ulazne matrice su kvadratne reda  $n$

pa ih dijelimo na isti način u blokove.

Dakle,  $p = q = r =$  (oznaka)  $= N$ , gdje je  $N$  tzv. “blok-red” matrice.

# Množenje matrica — blokovi (nastavak)

Podjela sve tri matrice  $A$ ,  $B$  i  $C$  ima isti oblik (napisan za  $C$ )

$$C = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1N} \\ C_{21} & C_{22} & \cdots & C_{2N} \\ \vdots & \vdots & \cdots & \vdots \\ C_{N1} & C_{N2} & \cdots & C_{NN} \end{bmatrix} .$$

Pojedini blokovi — podmatrice  $A_{ij}$ ,  $B_{ij}$  i  $C_{ij}$  su

• matrice istog tipa, označimo ga s  $n_i \times n_j$ .

Uočite da blokovi ne moraju više biti kvadratne matrice — općenito su pravokutne.

# Množenje matrica — blokovi (nastavak)

Za **veliĉine** blokova mora vrijediti

$$\sum_{i=1}^N n_i = n.$$

Kako se **određuju** veliĉine blokova  $n_i$ , za  $i = 1, \dots, N$  — malo kasnije.

Matriĉna operacija  $C = C + A * B$  sad ima “**blokovski**” oblik

$$C_{ij} = C_{ij} + \sum_{k=1}^N A_{ik} * B_{kj}, \quad i = 1, \dots, N, \quad j = 1, \dots, N.$$

# Množenje matrica — realizacija po blokovima

Programska realizacija na “blokovskoj” razini (po blokovima) ima ovaj opći oblik:

- 3 petlje po  $i$ ,  $j$ ,  $k$ , svaka od 1 do  $N$ ,
- operacija unutar tih petlji je

$$C_{ij} := C_{ij} + A_{ik} \cdot B_{kj},$$

tj. množenje i zbrajanje matrica.

Ova operacija ima isti oblik xGEMM kao i cijeli polazni problem (“rekurzija”), samo što matrice ne moraju biti kvadratne

$$(n_i \times n_j) = (n_i \times n_j) + (n_i \times n_k) * (n_k \times n_j).$$



## Blokovsko množenje matrica — petlje

Tri petlje za blokove smijemo permutirati — pa dobivamo 6 različitih varijanti blokovskog algoritma:

- $ijk, ikj, jik, jki, kij, kji$ .

Za “unutarnje” množenje pojedinih blokova, također, imamo odgovarajućih 6 varijanti osnovnog algoritma.

- Dakle, sve skupa, imamo 36 varijanti!

Tko hoće, neka proba sve. Ja neću.

U nastavku koristim

- istu varijantu (permutaciju petlji) i za blokovski i za osnovni (skalarni) algoritam.

# Blokovsko množenje matrica — veličine blokova

Ideja: veličine blokova izabrati tako da se unutarne množenje blokova

$$C_{ij} := C_{ij} + A_{ik} \cdot B_{kj}$$

(operacija xGEMM) obavlja u cacheu.

Postupak. Iz tablice brzina za odabrani osnovni algoritam

- nađemo približni maksimalni red  $n$  za koji još dobivamo punu “cache” brzinu.

Nazovimo taj red s  $n_{\text{cache}}$ .

## Veličine blokova (nastavak)

Cilj podjele na blokove je

- unutarnje množenje blokova mora raditi s matricama veličine manje (ili jednake)  $n_{\text{cache}}$ .

Dakle, mora vrijediti

$$n_i \leq n_{\text{cache}}, \quad i = 1, \dots, N.$$

Tome dodajemo raniji uvjet

$$\sum_{i=1}^N n_i = n.$$

Iz ovih uvjeta možemo odrediti broj blokova  $N$ .

## Broj blokova $N$

Uvrstimo  $n_i \leq n_{\text{cache}}$ , za  $i = 1, \dots, N$ , u relaciju za **zbroj**.

Izlazi

$$n = \sum_{i=1}^N n_i \leq N \cdot n_{\text{cache}},$$

ili

$$N \geq \frac{n}{n_{\text{cache}}}.$$

**Broj** blokova  $N$  mora biti **cijeli** broj i još (prirodno) želimo da

🔴  $N$  bude što **manji** — **najmanji** mogući!

Onda treba uzeti

$$N = \left\lceil \frac{n}{n_{\text{cache}}} \right\rceil = \left\lfloor \frac{n + n_{\text{cache}} - 1}{n_{\text{cache}}} \right\rfloor.$$

## Veličine blokova (nastavak)

Za nalaženje  $n_i$  standardno se koriste dva pristupa.

- “equal-sized” ili “uniform” — svi  $n_i$  imaju podjednaku veličinu  $n_i \approx n/N$ , tj. razlika među njima je najviše 1.
- “greedy” — svi  $n_i$  imaju maksimalnu veličinu  $n_{\text{cache}}$ , osim, eventualno, jednog od njih (prvi ili zadnji).

Ako želimo dobiti jednoznačnost rastava na blokove, zgodno je uzeti da su

- veličine blokova  $n_i$  sortirane — uzlazno ili silazno.

U nastavku uzimamo silazni poredak

$$n_1 \geq n_2 \geq \dots \geq n_N.$$

## “Equal-sized” — podjednake veličine blokova

Definiramo **ostatak**

$$n_r := n \bmod N.$$

**Podjela** na blokove izlazi iz **rastava** broja  $n$  oblika

$$n = \left\lfloor \frac{n}{N} \right\rfloor \cdot N + n_r = (N - n_r) \cdot \left\lfloor \frac{n}{N} \right\rfloor + n_r \cdot \left( \left\lfloor \frac{n}{N} \right\rfloor + 1 \right).$$

Veličine blokova  $n_i$  u **silaznom** poretku su

$$n_i = \begin{cases} \left\lfloor \frac{n}{N} \right\rfloor + 1, & \text{za } i = 1, \dots, n_r, \\ \left\lfloor \frac{n}{N} \right\rfloor, & \text{za } i = n_r + 1, \dots, N. \end{cases}$$

# “Greedy” — maksimalne veličine blokova

Definiramo **ostatak**

$$n_r := n \bmod n_{\text{cache}}.$$

**Podjela** na blokove izlazi iz **rastava** broja  $n$  oblika

$$n = \left\lfloor \frac{n}{n_{\text{cache}}} \right\rfloor \cdot n_{\text{cache}} + n_r.$$

Veličine blokova  $n_i$  u **silaznom** poretku su

$$n_i = n_{\text{cache}}, \quad i = 1, \dots, N - 1,$$
$$n_N = \begin{cases} n_{\text{cache}}, & \text{za } n_r = 0 \text{ (tj. } n_{\text{cache}} \text{ dijeli } n), \\ n_r, & \text{za } n_r > 0. \end{cases}$$

## Veličine blokova (nastavak)

U primjerima se koristi “equal-sized” ili “uniform” podjela.

Napomena. Pravu (najbolju) vrijednost za  $n_{\text{cache}}$  određujemo

• testiranjem blokovskog algoritma!

(Taj treba biti što brži.)



## *Blokovsko množenje matrica — indeksi*

I još, da nam se **indeksi** i **oznake** ne “pomiješaju”

• što indeksira **blokove**, a što **elemente**,

dodajemo **podindeks** “*b*” za sve što se odnosi na **blokove**.

## Blokovsko množenje matrica — primjer

IVF s *normal* opcijom za *jik* petlju daje brzine:

- 1050 MFlops za  $n \leq 50$ ,
- 205 MFlops za velike  $n$ .

IVF s *normal* opcijom za *jki* petlju daje brzine:

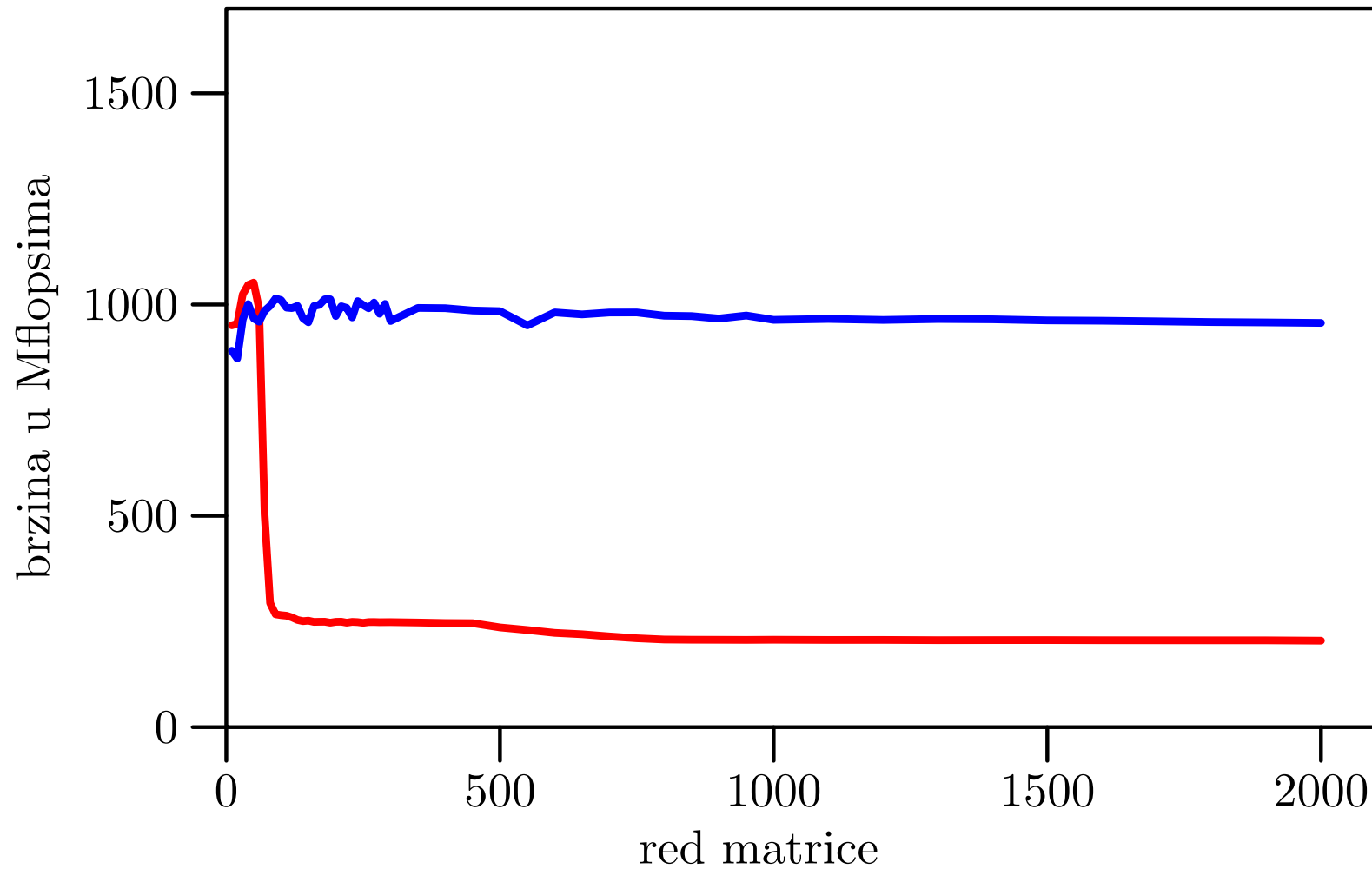
- 1100 MFlops za  $n \leq 300$ ,
- 840 MFlops za velike  $n$ .

IVF s *fast* opcijom za *jki* petlju daje brzine:

- 2000 MFlops za  $n \leq 300$ ,
- 1250 MFlops za velike  $n$ .

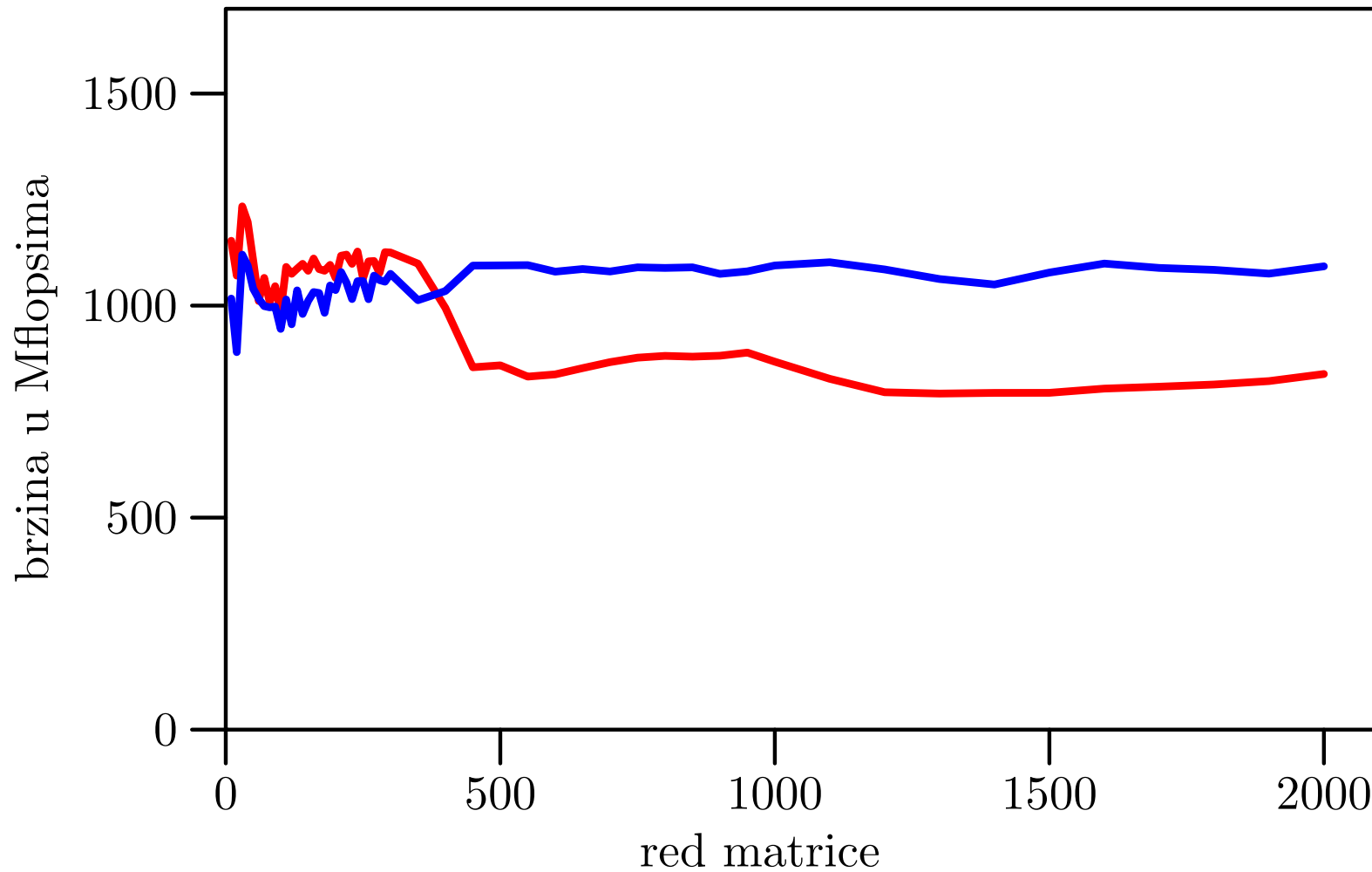
# BabyBlue, IVF, normal — jik *obični i blok (50)*

Pentium 4/660, 3.6 GHz, IVF, normal – Množ. mat. jik (50)



# BabyBlue, IVF, normal — jki *obični i blok (300)*

Pentium 4/660, 3.6 GHz, IVF, normal – Množ. mat. jki (300)



# BabyBlue, IVF, fast — jki *obični* i blok (300)

Pentium 4/660, 3.6 GHz, IVF, fast – Množ. mat. jki (300)

