

# *Parallelisierung der Gauß- Elimination mit CUDA*

Eine Einführung

vorgetragen von Christian Heinrich



# *Inhaltsverzeichnis*

1. Einführung in das Gauß-Verfahren
    - Funktionsweise des Algorithmus
    - Numerische Eigenschaften & Komplexität
    - Pivotisierung
    - Beispiel
  2. Parallelisierung mit CUDA
    - Was soll es bringen? (Konstruktions-)Ziele
    - Vereinbarungen
    - Funktionsweise
      - Schritte 1-5
    - Was „kostet“ es?
    - Was bringt es effektiv?
      - Geschwindigkeit & Tuning
  3. Fazit
  4. Quellen
- 
-

# 1. Einführung in das Gauß-Verfahren

- Motivation: Nicht-approximierte Lösung von lin. Gleichungssystemen  $Ax = b$  ( $x$  wird gesucht)
  - Aber: Nur endliche Genauigkeit
  - Für Algo werden nur „Elementaroperationen“ verwendet, die die Lösung nicht verändern:
    - Multiplikation einer Zeile mit einem Skalar  $\neq 0$
    - Addition einer Zeile zu einer anderen
    - (Vertauschen zweier Zeilen)
  - Verfahren: Bringe Matrix auf Dreiecksgestalt, indem schrittweise in jeder Spalte alle Elemente unter der Hauptdiagonalen mittels Elementaroperationen auf 0 gebracht werden
- 
-

# 1. Einführung in das Gauß-Verfahren

- Korrektheit: Skizze: Vgl. Funktionsweise
  - Komplexität:  $O(n^3)$
  - „Alles oder nichts“: Bei Abbruch vor Terminierung KEINE Näherungslösung (-> nicht iterativ)
  - Numerische Stabilität: Es gibt keine Pivotstrategie, für die die Gauß-Elimination garantiert num. stabil ausgeht; Wahrscheinlichkeit, auf Matrix zu treffen, für die die Gauß-El. instabil ausgeht, ist äußerst gering
  - Speicherung: Speichere anstelle der 0-en in der unteren Dreiecksmatrix die entsprechenden Zeilenfaktoren
- 
-

# *1. Einführung in das Gauß-Verfahren*

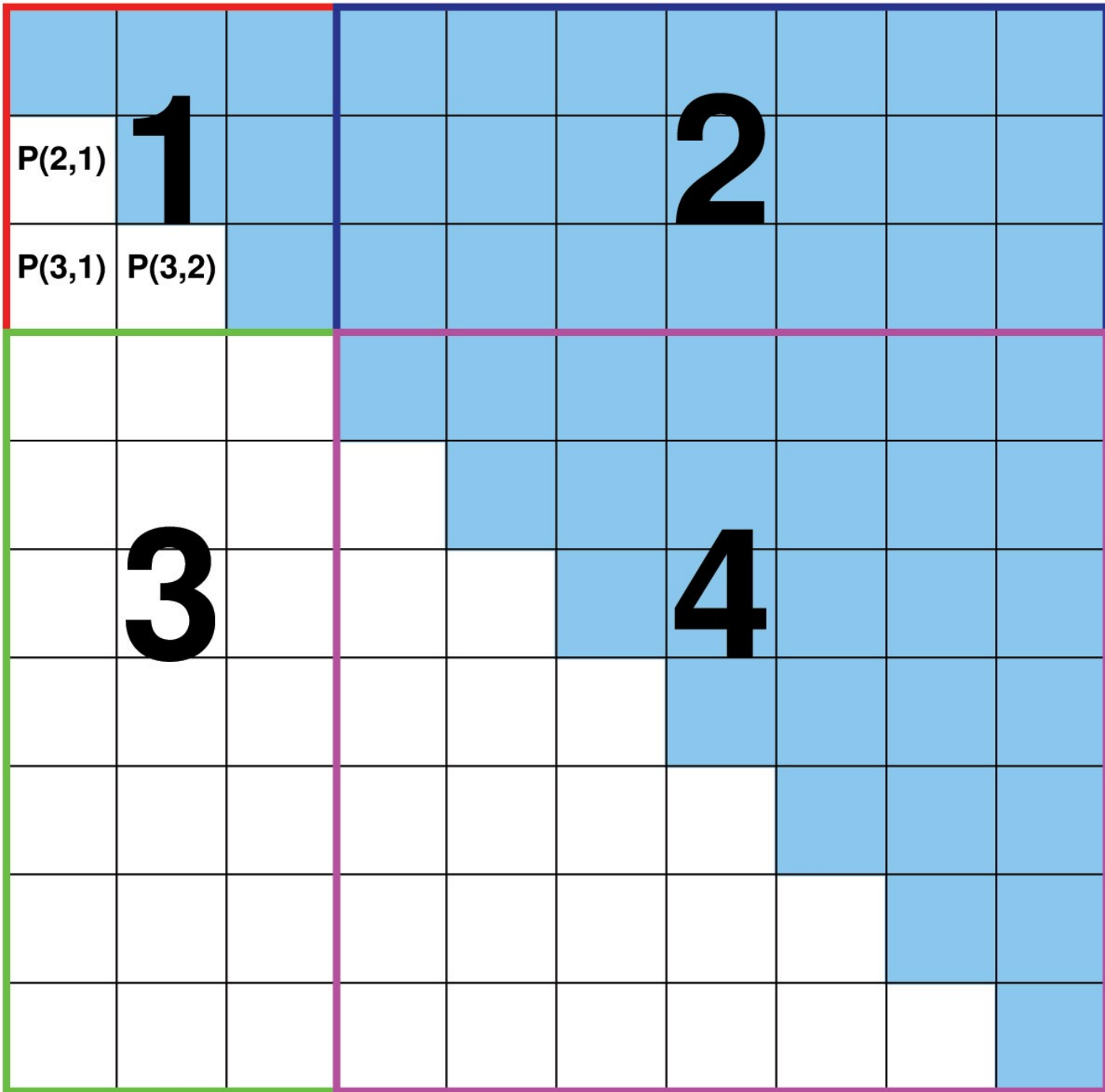
- Pivotisierung:
    - Geschicktes Wählen von Elementen, mit denen die Spaltenoperationen durchgeführt werden
    - Hilft, falls ein Diagonalelement frühzeitig in nichtsingulärer Matrix verschwindet
- 
-

## 2. Parallelisierung mit CUDA: (Konstruktions-)Ziele

- Parallelisierung soll helfen, den Algo (zeitlich) schneller abzuarbeiten, denn:
  - Viele Schritte können unabh. voneinander durchgeführt werden: z.B.
    - Multipl. der Spalten einer Zeile mit einem Skalar
    - Addition zweier Zeilen (spaltenweise unabh.)
  - Massive Nutzung des SharedMemory soll Zugriffe auf Hauptspeicher minimieren; Threads sollen sich nicht gegenseitig ausbremsen
- 
-

## 2. Parallelisierung mit CUDA: Vereinbarungen

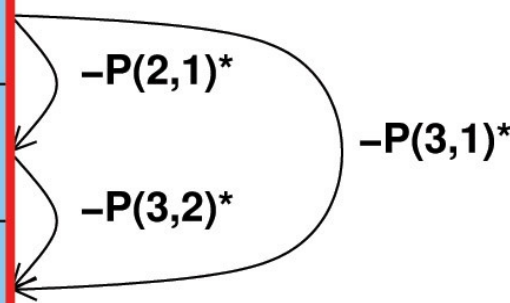
- Im Folgenden:
    - $S$  bezeichne eine natürliche Zahl
    - Speicheralignment: Vorteil, falls  $S$  2er Potenz
    - Vorerst:  $S = 16$
    - $N$  bezeichne eine natürliche Zahl;  $\mathbb{C}$  sei  $N$  Vielfaches von  $S$
    - $A$  bezeichne eine quadratische  $N \times N$  Matrix
    - $A$  sei stets diagonaldominant; das erspart uns die Spaltenpivotsuche
    - $A^{S(i,j)}$  sei der  $S \times S$  Block in  $A$ , der  $a_{ij}$  als linkes oberes Element enthält
    - $A$  hat also je  $N/S$  Zeilen & Spalten von Blöcken
- 
-



**A**

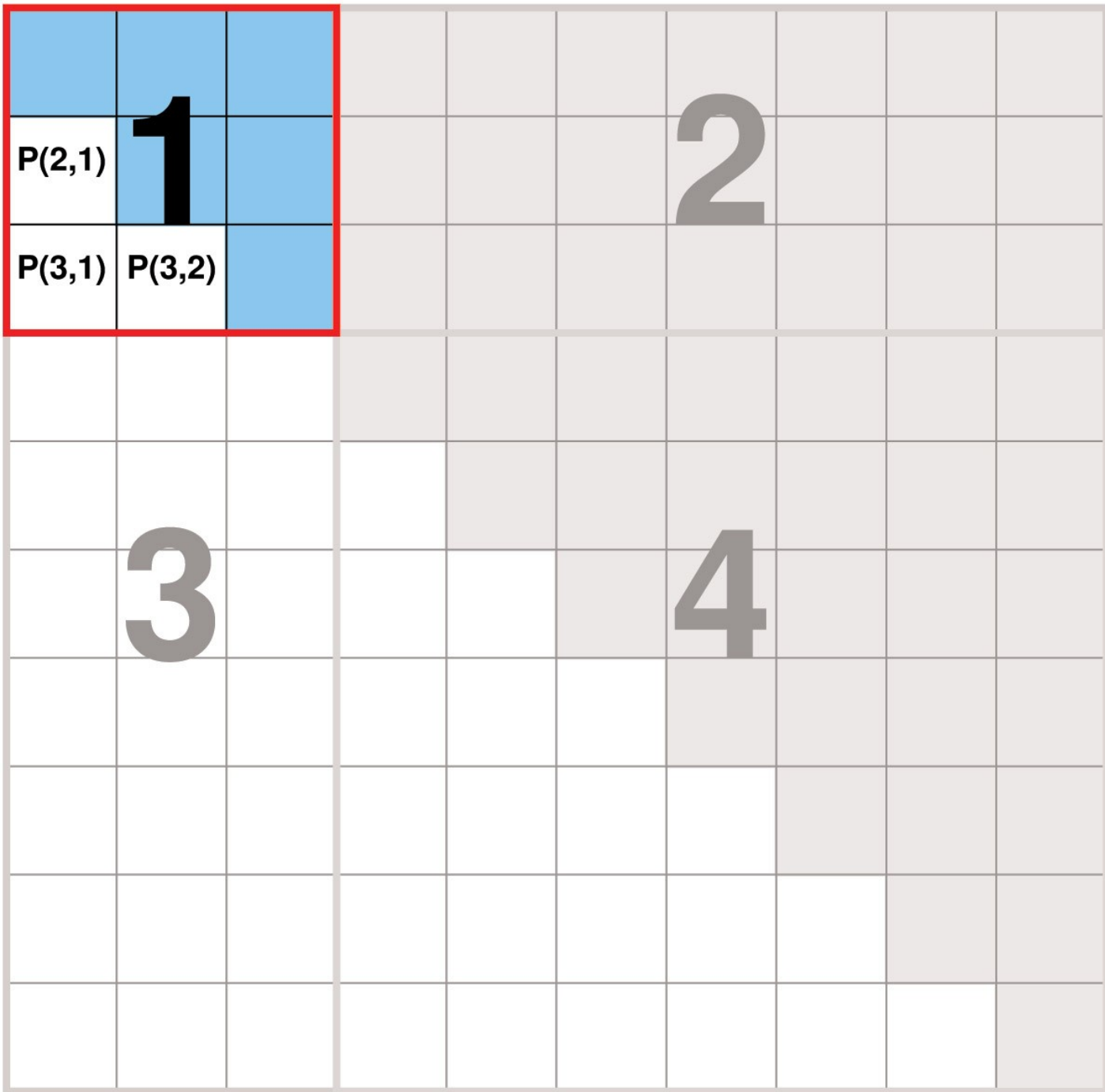


**b**

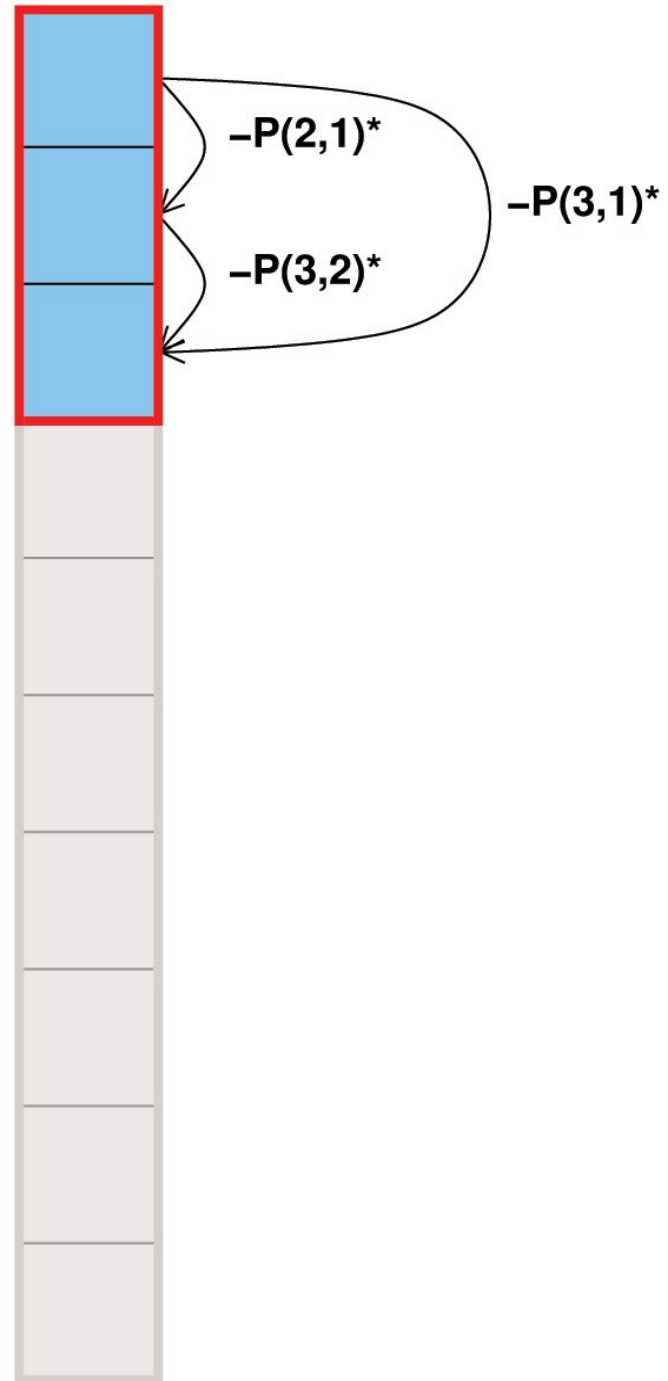


## ***2. Parallelisierung mit CUDA*** ***Funktionsweise***

1. G.-El. auf linken oberen  $S \times S$  Block anwenden. Auch  $b$  wird bearbeitet
  2. Zeilenoperationen auf ersten  $S$  Zeilen nachholen
  3. Ersten  $S$  Spalten unterhalb d. Diag. = 0 bekommen. Bearbeite auch  $b$ .
  4. Spalten- & Zeilenoper. auf Hauptteil nachholen
  5.  $i$ -te Rekursion auf Matrix der Größe  $(N-i \times S) \times (N-i \times S)$  aufrufen
- 
-



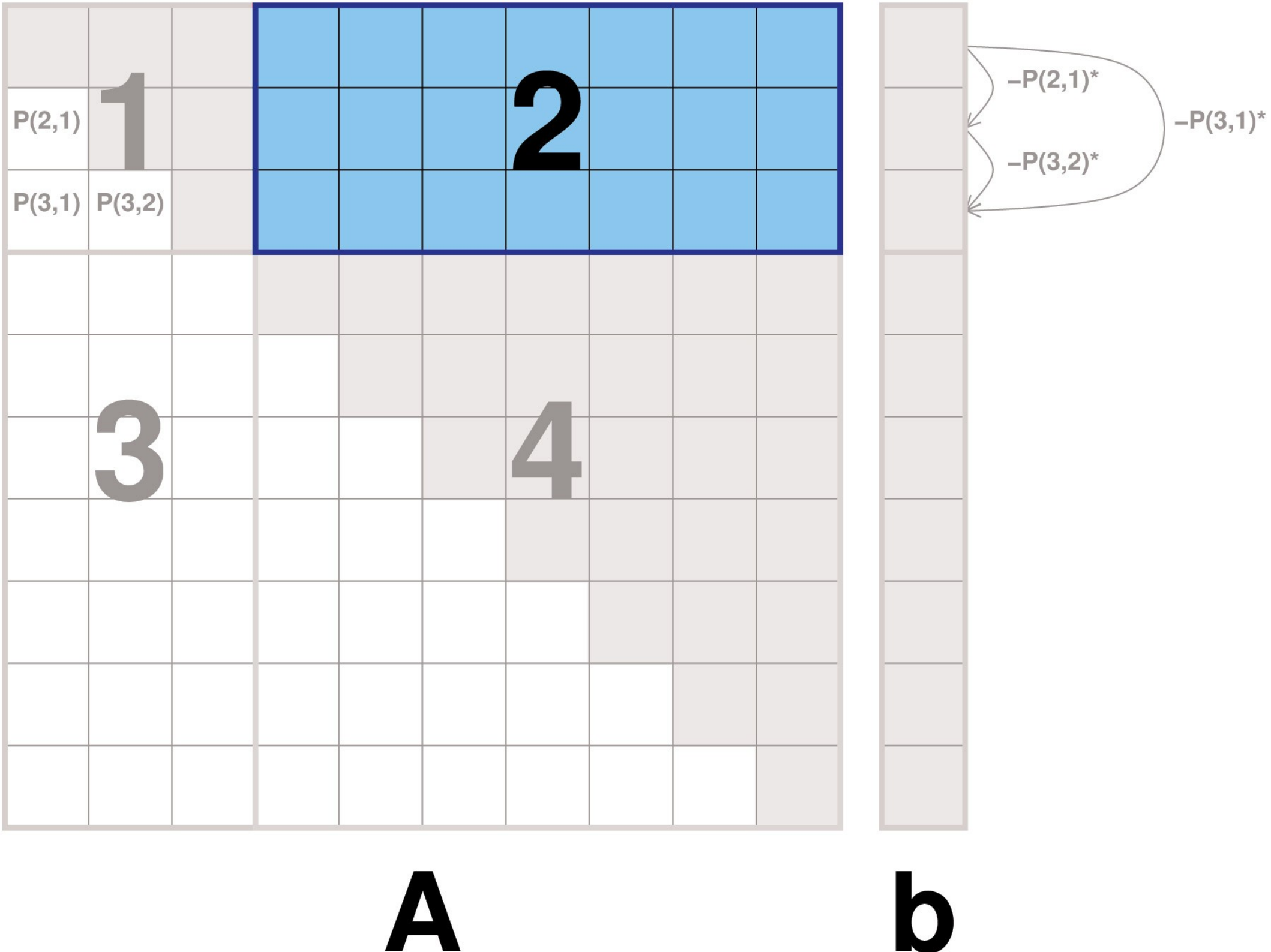
**A**



**b**

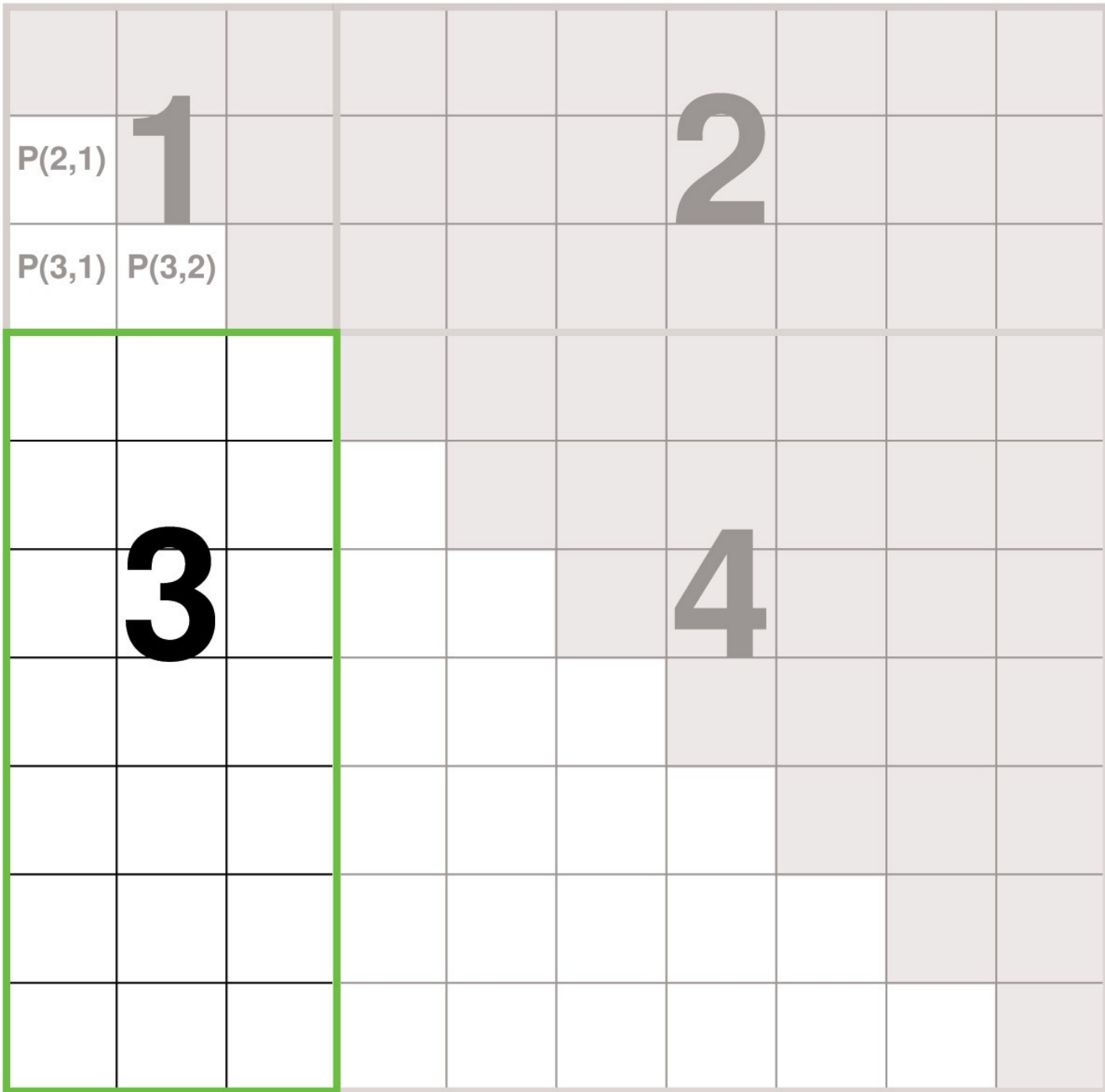
## ***2. Parallelisierung mit CUDA:*** ***1. Schritt***

1. Grundsätzlich: Jeder Koeffizient hat seinen eigenen Thread
  2. Führe normale Gauß-Elimination im linken oberen  $S \times S$  Block durch
  3. Speichere die Zeilenfaktoren wie besprochen
  4. Bearbeite auch  $b$
- 
-

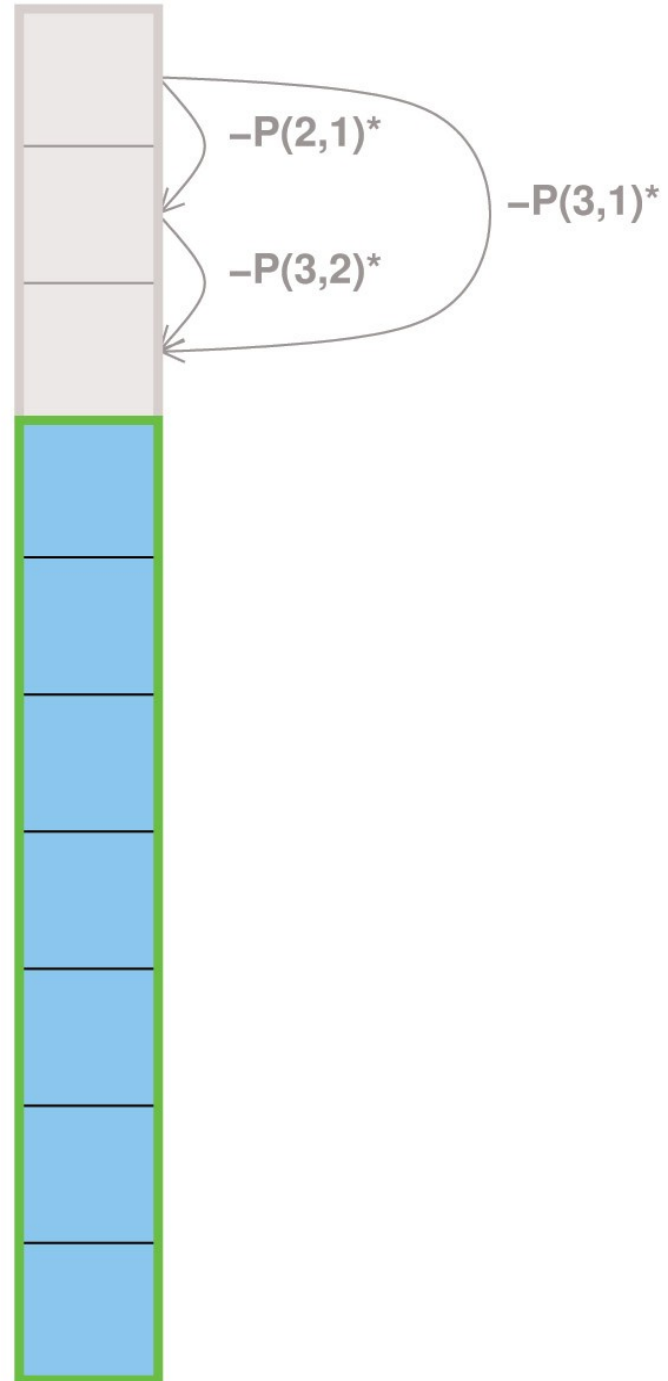


## ***2. Parallelisierung mit CUDA:*** **2. Schritt**

1. Bearbeite die restlichen  $S$  ersten Zeilen
    - Spaltenweise unabh., da Zeilenfaktoren bereits vorhanden -> ins SharedMemory kopieren
    - Bearbeitung auch hier in  $S \times S$  Blöcken
    - Pro Block: Zeilenoperationen voneinander abhängig!
  2. Aktuellen Block ins SharedMemory kopieren
  3. Zeilen entsprechend Zeilenfaktoren abziehen
  4. Block zurück in den Hauptspeicher schreiben
- 
-



**A**

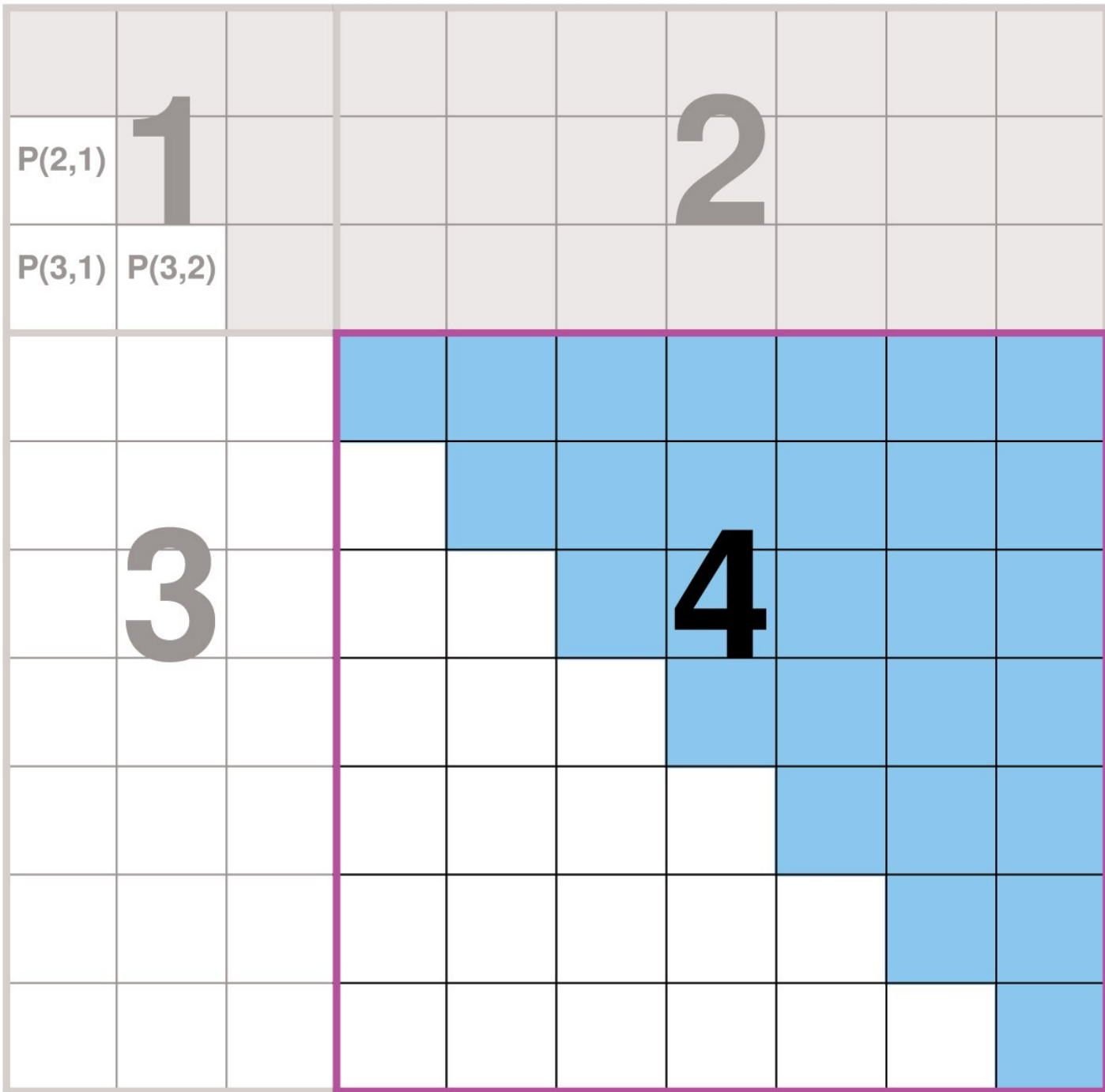


**b**

## **2. Parallelisierung mit CUDA:**

### **3. Schritt**

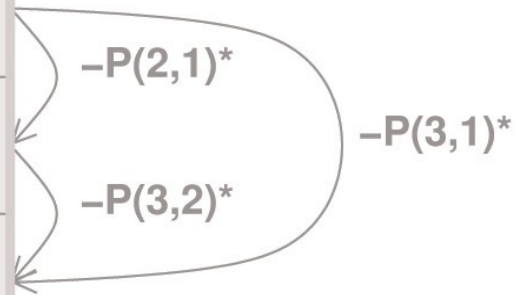
- Erweitere nun Zeilenoperationen auf alle  $S$  ersten Spalten, d.h. bringe diese Spalten unterhalb der Diagonalen auf 0
  - Geht zeilenweise unabh.
  - Vorgehensweise: Wieder Blöcke  $A^S(0,0)$  und  $A^S(i,0)$  laden und Schritte durchführen; wg. Spaltenabhängigkeit aber häufiges synchronisieren notwendig!
  - Parallelisierung sonst wie in 2., insb. sinnvoll, mehrere Blöcke von einem Thread-Block bearbeiten zu lassen
- 
-



**A**



**b**



## ***2. Parallelisierung mit CUDA:***

### ***4. Schritt, (1)***

- Schritt 4 kümmert sich um den Hauptteil und braucht die meiste Rechenzeit
  - Aber: Alle benötigten Zeilenfaktoren stehen zur Verfügung, d.h. kaum Synchronisation notwendig; pro zu bearbeitendem Block werden 2 Blöcke (Zeilenfaktoren und Pivotzeilen) benötigt
  - Zeilen- und Spaltenweise unabh.
  - Wichtig, mehrere Blöcke zusammenzufassen ( $\sim 2 \times 2$ ), sonst große Einbußen bei Speichergeschw.!
- 
-

## ***2. Parallelisierung mit CUDA:***

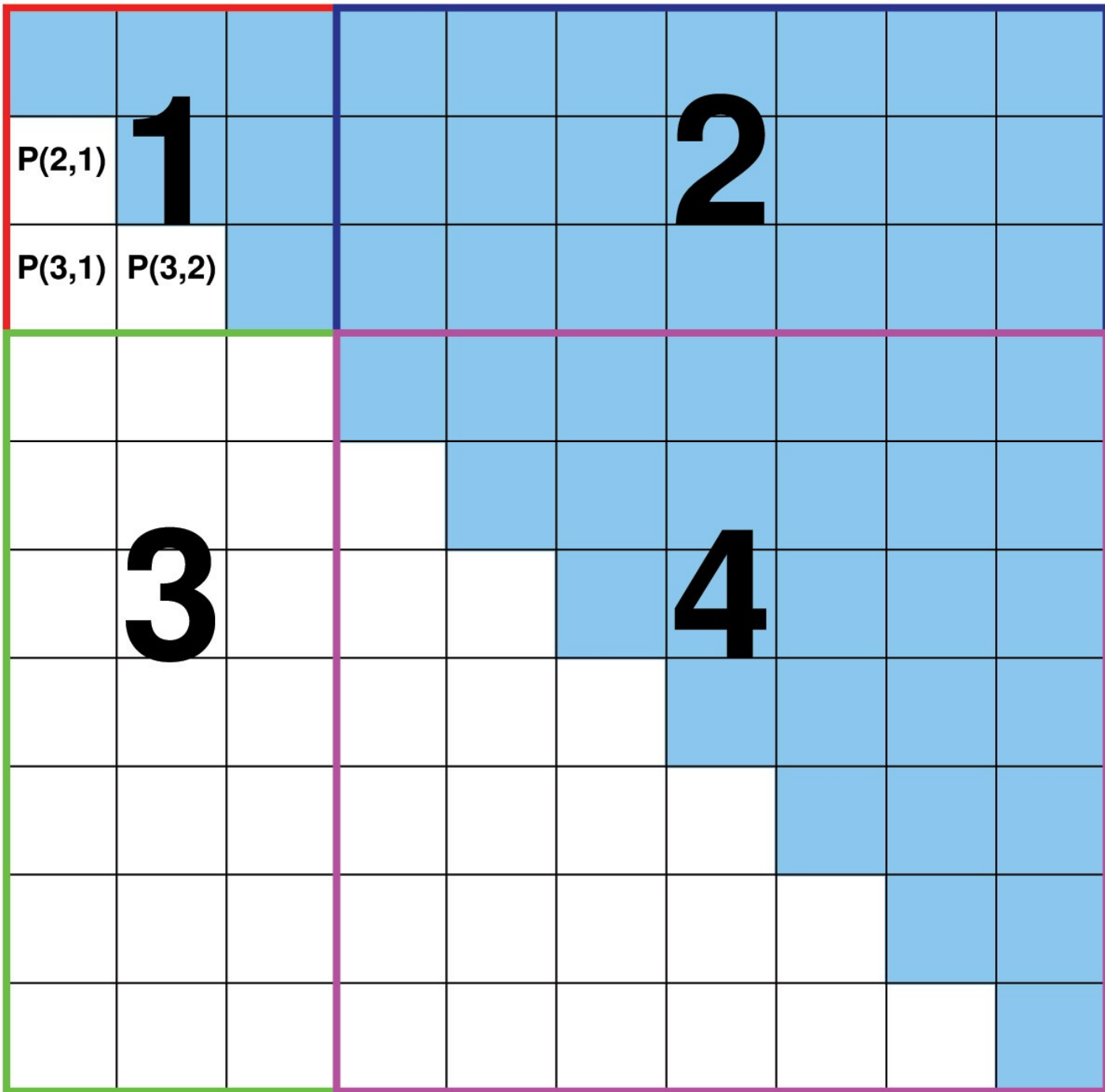
### ***4. Schritt, (2)***

- S Zeilen werden gleichzeitig abgezogen
  - Keine Synchronisation mit umliegenden Koeff.
  - Speicherzugriffe pro Koeff.:
    - 2x Hauptspeicher (1x lesen, 1x schreiben)
    - 2\*S SharedMemory ReadOnly (-> folgt aus S multiply-add-Operationen)
    - Zusätzlich 2/m Operationen vom Zusammenfassen
- 
-

## *2. Parallelisierung mit CUDA:*

### *5. Schritt*

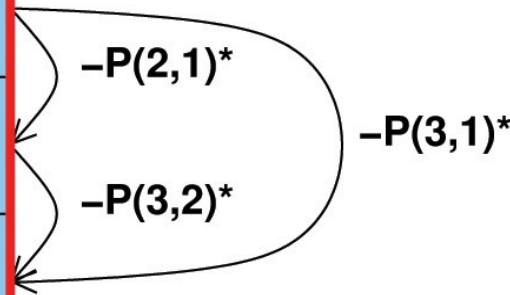
- Aufruf der  $i$ -ten Rekursion auf Restmatrix der Größe  $(N-i*S) \times (N-i*S)$



**A**



**b**



## 2. Parallelisierung mit CUDA: Kernel für Schritt 1

```
// Hier: Ohne Vektor b
procedure eliminate_first_block(Matrix A)
begin
    col           := threadIdx.x;           // x Koordinate des Threads
    row           := threadIdx.y;           // y Koordinate des Threads

    // Dieser Thread bearbeitet sharedBlock.Element(row, col)
    sharedBlock.Element(row, col) = A.element(row, col); // A ins SharedMemory kopieren
    __synchronize(); // Warten, dass alles im SharedMemory steht

    for ( r := 0; r < S-1; ++r ) // r entspricht der Spalte, die jetzt bearbeitet wird
    do
        // Zeilenfaktor berechnen, um diese Spalte 0 zu kriegen
        rowfactor := sharedBlock.Element(row, r) / sharedBlock.getElement(r, r);
        __synchronize(); // Alle Zeilenf. müssen berechnet sein, bevor überschrieben wird

        if ( row > r ) // Zeile liegt unter der aktuell zu bearbeitenden Zeile?
        Then
            if ( col > r ) // Element liegt rechts von aktueller Spalte -> Zeilen abziehen
                sharedBlock.Element(row, col) -= rowfactor*sharedBlock.Element(r, col);
            if ( col == r ) // Element liegt in gerade zu bearbeitender Spalte -> Zeilenfaktor speichern
                sharedBlock.Element(row, col) := rowfactor;
        fi;

        __synchronize(); // Alle Operationen erst beenden lassen, dann weitermachen
    od;
    A.element(row, col) := sharedBlock.Element(row, col); // Schreibe Ergebnisse zurück in den Hauptspeicher
end;
```

## 2. Parallelisierung mit CUDA

### Kosten

- Materialkosten: CUDA kompatible Grafikkarte (~ \$1000)
  - Ressourcenkosten: Strom (~ 200 W)
  - Personalkosten:
    - Zusätzlicher Programmieraufwand
    - Erhöhte Fehleranfälligkeit
    - Ggf. schwierigeres Debugging
  - Performancekosten:
    - Threads haben oft Abhängigkeiten, weshalb teilweise häufig synchronisiert werden muss -> Trotzdem viel schneller als auf 1 CPU
- 
-

## *2. Parallelisierung mit CUDA: Was bringt es effektiv?*

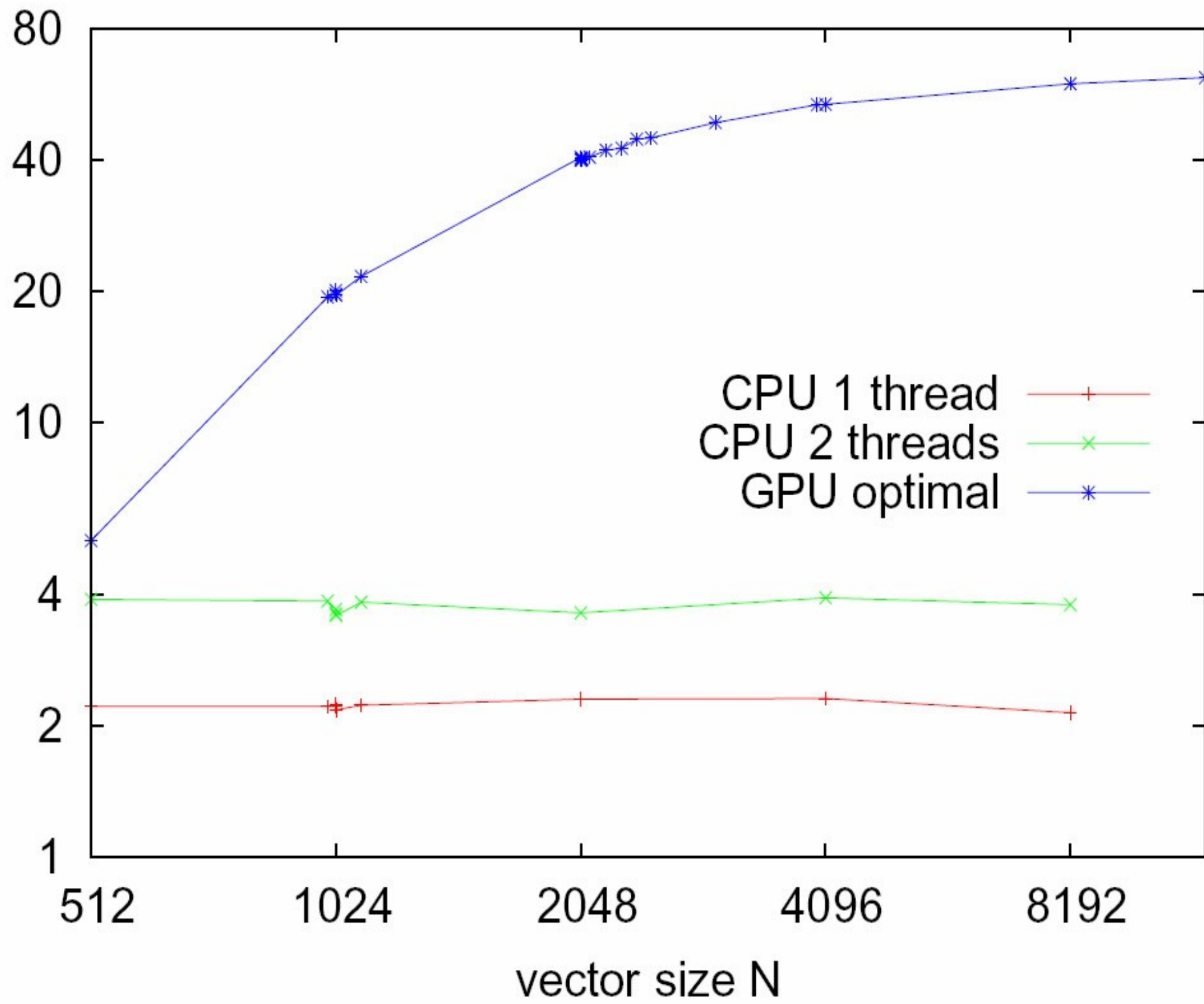
- Vergleich:

GeForce 8800 Ultra (16 Kerne)

vs

Intel Core2Duo E6750 (2 Kerne)

GFlops



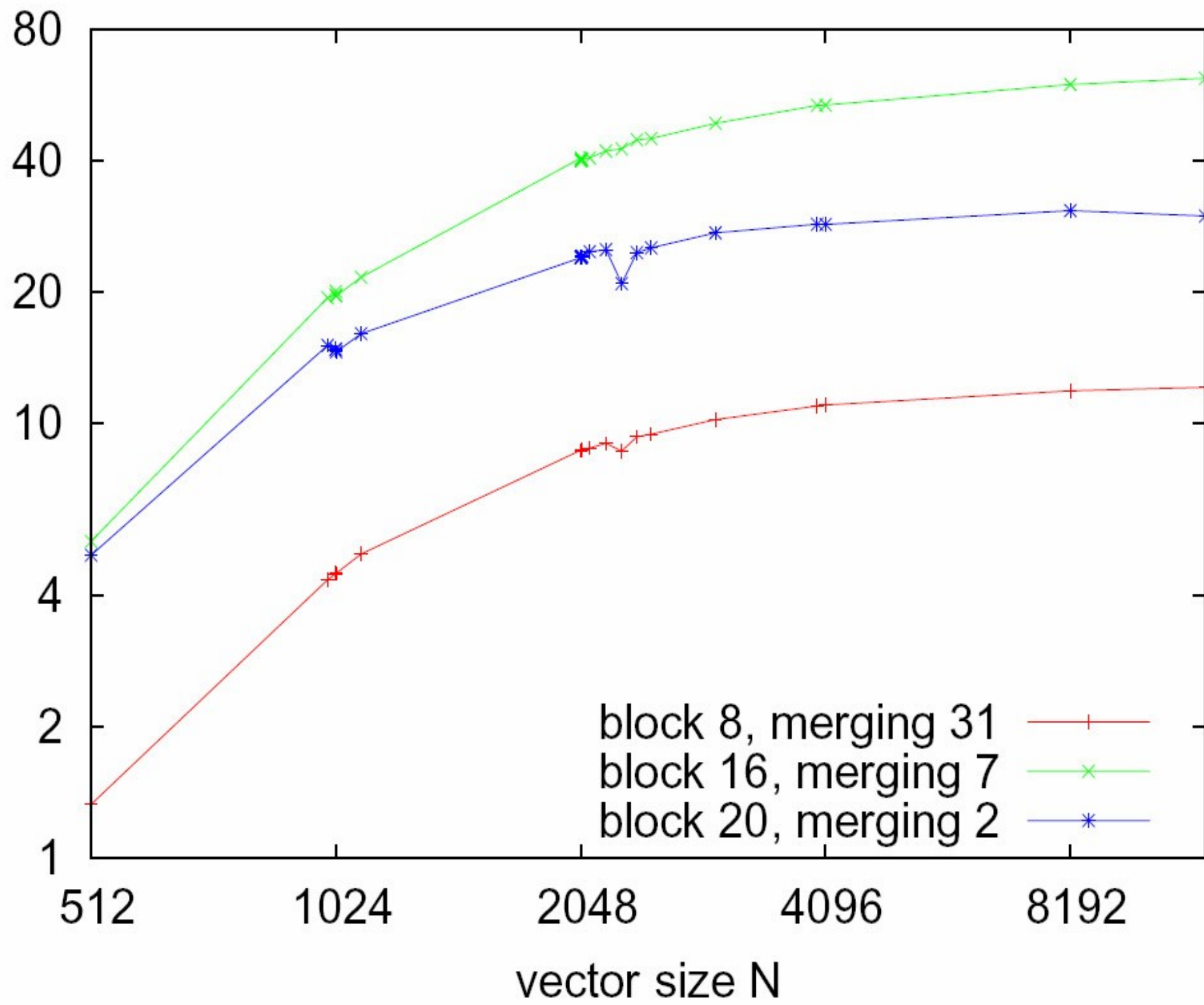
## ***2. Parallelisierung mit CUDA: Was bringt es effektiv?***

- Die CPU hat bei doppelten Kernen nahezu doppelte Performance (~ Faktor 1,8)
  - Aber: Dieser Faktor kommt nur zustande, weil Speicherbus nicht ausgelastet; geht nicht immer so weiter!
  - Trotzdem ist die GPU nahezu 20mal schneller als die CPU mit 2 Kernen, bei nahezu gleichem Preis & Stromverbrauch
- 
-

## *2. Parallelisierung mit CUDA: Tuning? (1)*

- In dieser Implementation ist z.B. die Blockgröße  $S$  veränderbar; bringt aber nicht viel, da bei  $S = 8$  zuviele Speicherzugriffe und bei  $S = 20$  gibt es Probleme bei Abhängigkeiten von Lesezugriffen

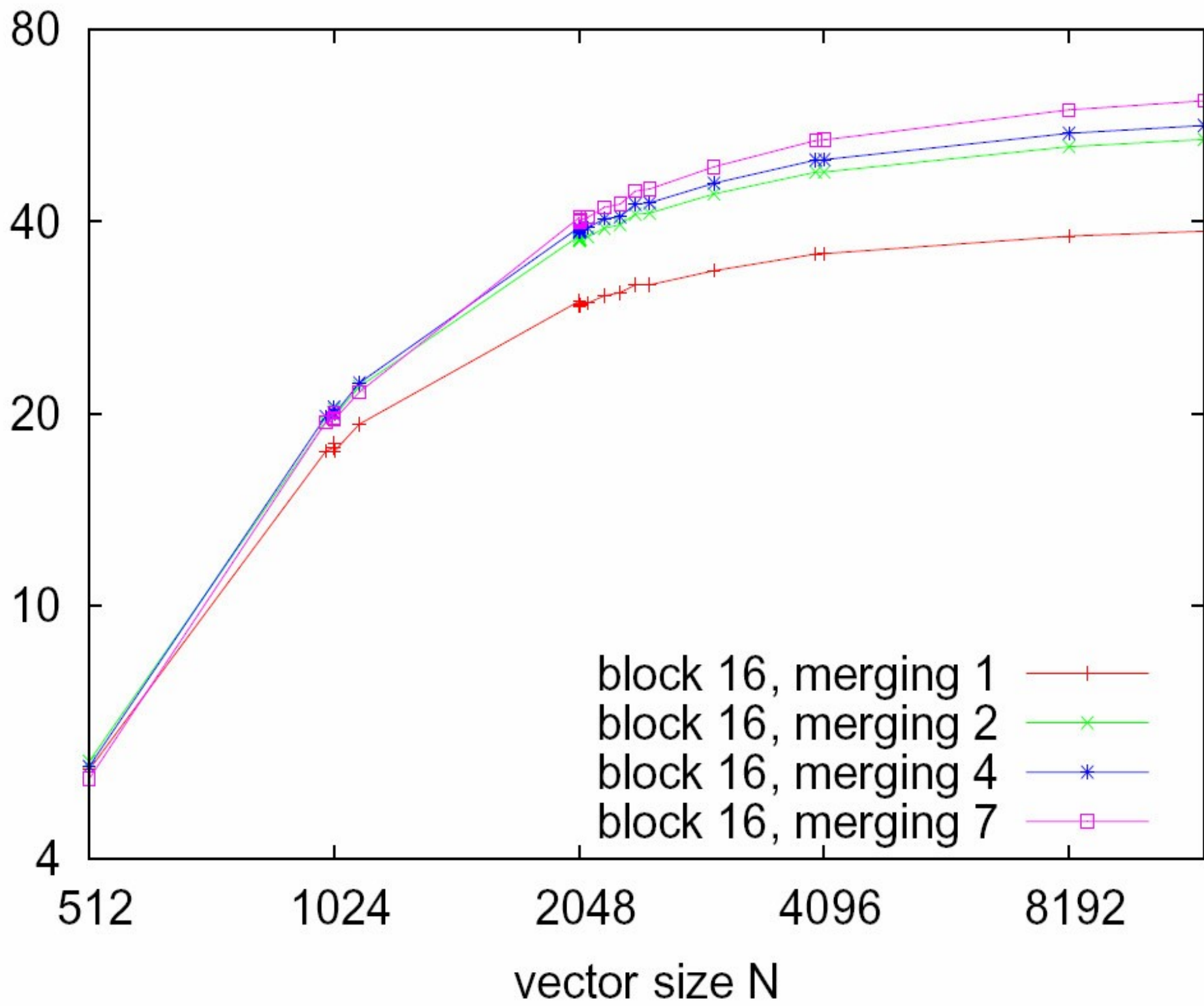
GFlops



## *2. Parallelisierung mit CUDA: Tuning? (2)*

- Man könnte sonst in Schritt 4 auch mehr Blöcke zusammenfassen;
    - 2 x 2 ist sinnvoll
    - Tests zeigen: Mehr bringt nicht viel, da vorwiegend im SharedMemory gerechnet wird (Selbst bei 50% Speichertakt fällt die Performance um nur 10%!)
- 
-

GFlops



## 3. Fazit

- Die GPU spielt deutlich ihre Stärken aus; der CPU-Prozessor hat hier keine Chance
  - Effiziente Blockgrößen sind unerlässlich
  - Unschön: Nur für – in Berechnungen – sehr kleine  $n$  machbar:  $0 \leq n \leq 12.000$ , da hier Speicher effektiv limitierend
  - Offene Fragen: Wie steigt die Performance, wenn mehrere Grafikkarten im Verbund arbeiten?  
Proportional zur Anzahl der Grafikkarten oder ist irgendwann Obergrenze erreicht?
- 
-

## 3. Quellen

- Dokumentation: Blockweise Gauß-Elimination ohne Pivotisierung, (Dr. Axel Arnold, Fraunhofer Institut, 2008)
  - Implementation: Dr. Axel Arnold
  - Illustrationen: Dr. Axel Arnold (Freistellung / Bearbeitung durch Christian Wolf)
  - Nvidia CUDA-Guide
  - Numerik-I-Skript (Prof. Dr. R. Seydel, SS 2007)
- 
-