› Programmierbeispiele und Implementierung

Name: Michel Steuwer | E-Mail: michel.steuwer@wwu.de

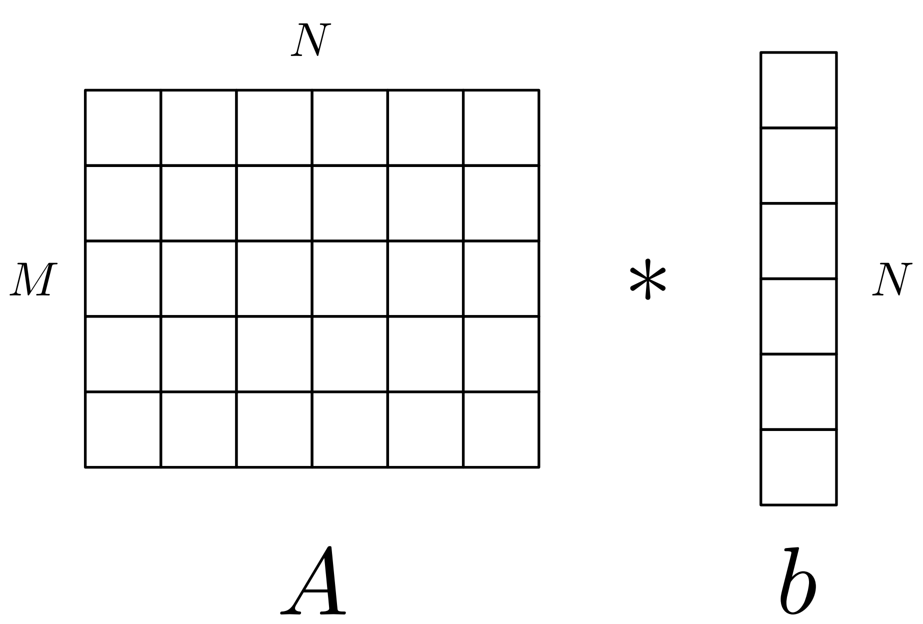
› Übersicht

- › Matrix Vektor Multiplikation
- › Mandelbrotmenge / Apfelmännchen berechnen
- › Kantendetektion mit dem Sobel Operator

wissen.leben
WWU Münster

› Matrix-Vektor-Multiplikation: Aufteilen der Berechnung

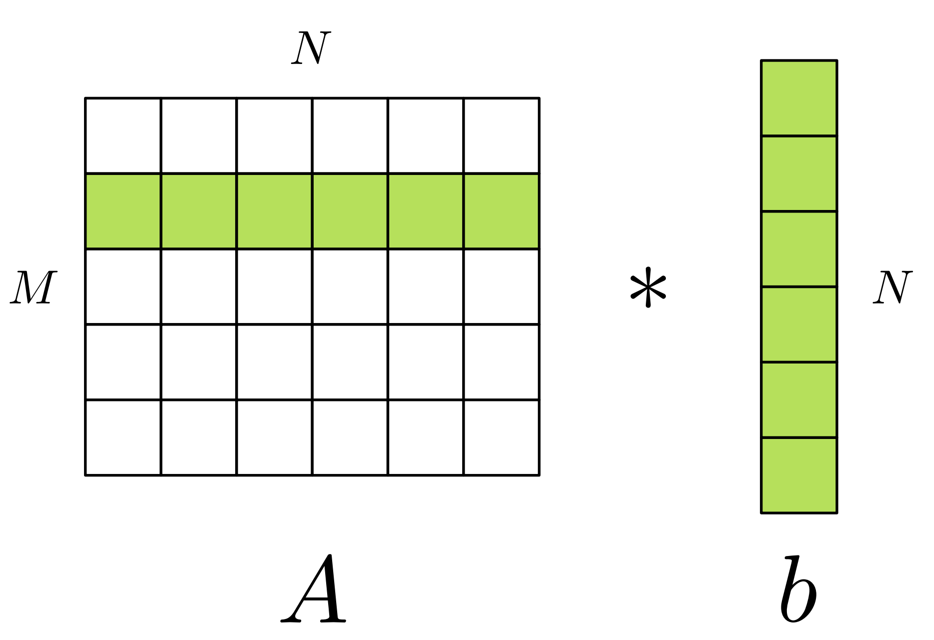
- › Die Berechnung der einzelnen Elemente des Ergebnis-Vektors sind unabhängig voneinander
- › Daher berechnet je ein Block das Skalarprodukt einer Matrix-Zeile mit dem Vektor b



wissen.leben WWU Münster

› Matrix-Vektor-Multiplikation: Aufteilen der Berechnung

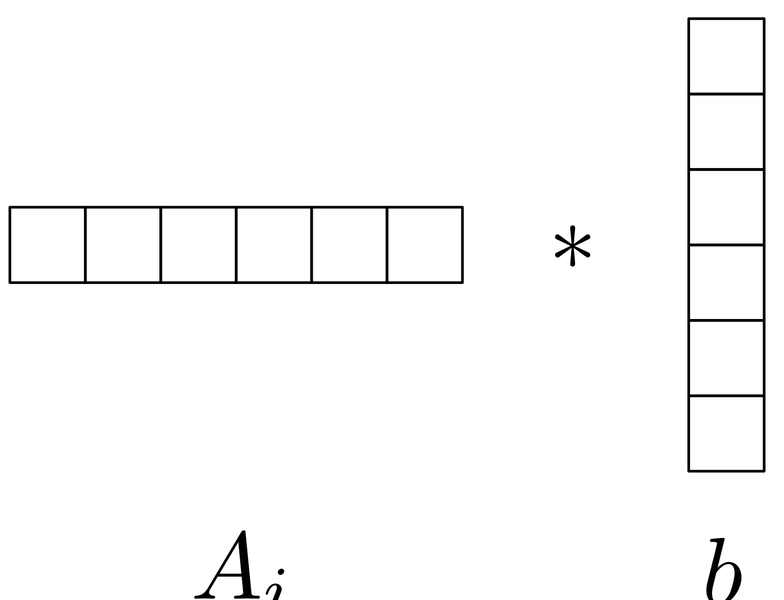
- Die Berechnung der einzelnen Elemente des Ergebnis-Vektors sind unabhängig voneinander
- Daher berechnet je ein Block das Skalarprodukt einer Matrix-Zeile mit dem Vektor b



wissen.leben
WWU Münster

› Aufteilen der Berechnung

- Die Berechnung des Skalarprodukts einer Matrix-Zeile A_i mit dem Vektor b lässt sich parallelisieren
- Jeder Thread innerhalb eines Blocks berechnet einen Teil des Skalarprodukts anschließend müssen alle Zwischenergebnisse aufsummiert werden



wissen.leben WWU Münster

› CUDA Kernel: Matrix Vektor Multiplikation

```
1  __global__ void matrixVectorMul(float* c, float* A, float* b, int m, int n) {
2       __shared__ float data[BLOCK_SIZE];
3       int size = (n / BLOCK_SIZE);
4       if ( (n % BLOCK_SIZE) != 0 )
5            size += 1;
6
7       data[threadID.x] = calculatePart(A, b, size, n);
8
9       __syncthreads();
10
11      sumUpAllParts( data, size );
12
13      if (threadID.x == 0)
14           c[blockID.x] = data[0];
15 }
```

wissen.leben
WWU Münster

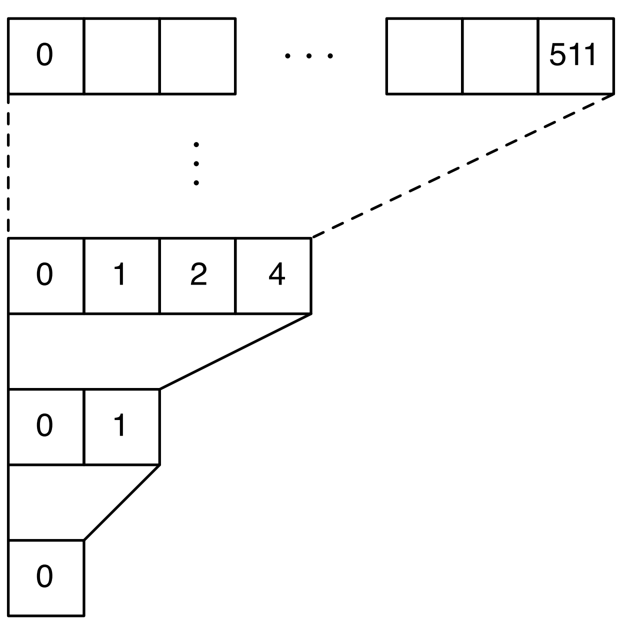
› CUDA Kernel: Berechnung eines Teils

```
1  __device__ float calculatePart( float* A, float* b, int size, int n ) {
2      float result;
3
4      for ( int i  = threadID.x * size;
5                i  < (threadID.x + 1) * size;
6                i += 1 ) {
7           if (i < n)
8                result += A[blockID.x * n + i] * b[i];
9      }
10
11     return result;
12 }
```

wissen.leben WWU Münster

› Zwischenergebnisse aufsummieren

- › Zum Aufsummieren der Zwischenergebnisse wird schrittweise die Zahl der rechnenden Threads halbiert
- › Die verbleibenden Threads addieren je zwei Zwischenergebnisse auf
- › Zum Schluss addiert der letzte verbleibende Thread die letzten beiden Zwischenergebnisse auf und erhält so das Resultat



wissen.leben WWU Münster

› CUDA Kernel: Zwischenergebnisse aufsummieren

```
1  __device__ void sumUpAllParts( float* data ) {
2      __shared__ float tempData[BLOCK_SIZE/2];
3
4      int thread_count = BLOCK_SIZE;
5      while (thread_count > 1) {
6          if ( threadID.x < (thread_count / 2) )
7              tempData[threadID.x] =    data[threadID.x]
8                                      + data[threadID.x + (thread_count/2)];
9
10          thread_count = thread_count / 2;
11
12         data[threadID.x] = tempData[threadID.x];
13
14         __syncthreads();
15     }
16 }
```

wissen.leben
WWU Münster

› CUDA Kernel: Matrix Vektor Multiplikation

```
1  __global__ void matrixVectorMul(float* c, float* A, float* b, int m, int n) {
2       __shared__ float data[BLOCK_SIZE];
3       int size = (n / BLOCK_SIZE);
4       if ( (n % BLOCK_SIZE) != 0 )
5            size += 1;
6
7       data[threadID.x] = calculatePart(A, b, size, n);
8
9       __syncthreads();
10
11      sumUpAllParts( data, size );
12
13      if (threadID.x == 0)
14           c[blockID.x] = data[0];
15 }
```

wissen.leben WWU Münster

› Host Code: Speicherverwaltung

```
1  ...
2  // allocate device memory for matrix A and vector b
3  float* device_A;
4  cudaMalloc( (void**) &device_A, sizeof(float) * M * N );
5  float* device_b;
6  cudaMalloc( (void**) &device_b, sizeof(float) * N );
7
8  // copy host memory to device (both matrix A and vector v)
9  cudaMemcpy(device_A, host_A, sizeof(float) * M * N, cudaMemcpyHostToDevice);
10 cudaMemcpy(device_b, host_b, sizeof(float) * N, cudaMemcpyHostToDevice);
11
12 // allocate device memory for result vector c
13 float* device_c;
14 cudaMalloc( (void**) &device_c, sizeof(float) * M );
15 ...
```

wissen.leben WWU Münster

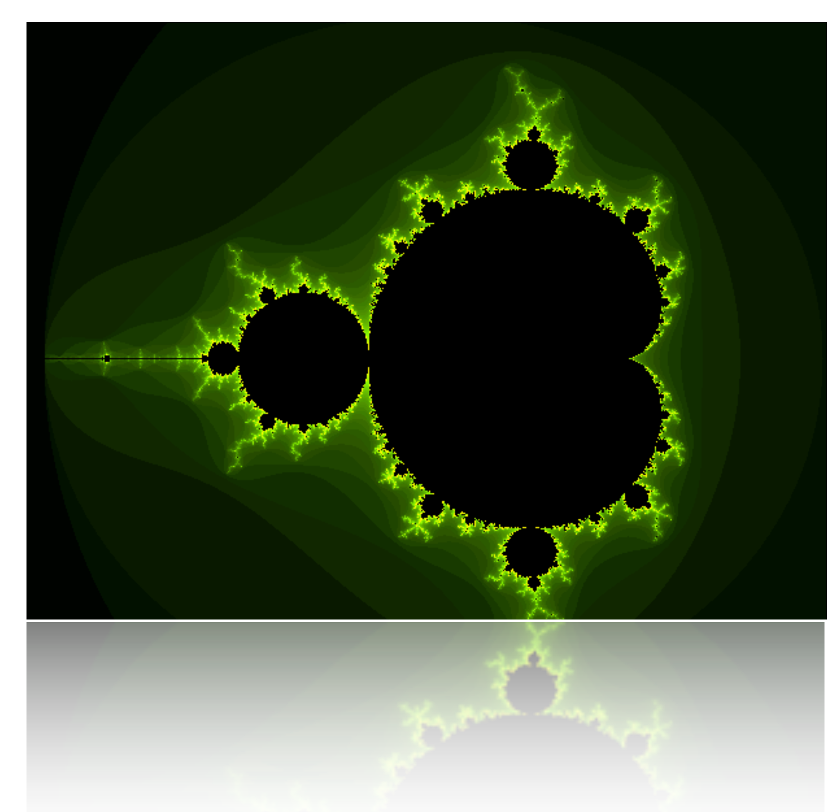
› Host Code: Aufruf des CUDA Kernels

```
1  ...
2  // setup execution parameters and execute the kernel
3  dim3 blockSize(BLOCK_SIZE);
4  dim3 gridSize(M);
5  matrixVectorMul<<<gridSize, blockSize>>>(device_c, device_A, device_b, M, N);
6
7  // copy result from device to host
8  cudaMemcpy(host_c, device_c, sizeof(float) * M, cudaMemcpyDeviceToHost);
9
10 // free device memory
11 cudaFree(device_A);
12 cudaFree(device_b);
13 cudaFree(device_c);
14 ...
```

wissen.leben
WWU Münster

› Mandelbrotmenge / Apfelmännchen berechnen

- Die Mandelbrotmenge ist eine selbstähnliche Menge in den komplexen Zahlen
- Die Mandelbrotmenge kann visualisiert werden, dabei sind die Berechnungen der Pixels des Ergebnisbildes unabhängig voneinander
- Für jeden Pixel wird eine Iteration durchgeführt.
  Abhängig von der Anzahl der Iterationsschritte wird der Pixel eingefärbt.



wissen.leben WWU Münster

› CUDA Kernel: Mandelbrotmenge berechnen

```
1  __global__ void MandelbrotSet(unsigned char* image, float startX, float startY,
2                                float dx, float dy, int width, int height) {
3      int pixelX = blockIdx.x * blockDim.x + threadIdx.x;
4      int pixelY = blockIdx.y * blockDim.y + threadIdx.y;
5
6      if (pixelX >= width || pixelY >= height)
7          return;
8      float x = startX + pixelX * dx;
9      float y = startY + pixelY * dy;
10
11     int n = Iterate( x, y );
12
13     SetPixelColor( image, pixelX, pixelY, width, height, n );
14 }
```

wissen.leben WWU Münster

› Host Code: CUDA und OpenGL

```
1  void display(void)
2  {
3      unsigned char* image = NULL;
4      cudaGLMapBufferObject((void**)&image, buffer);
5
6      dim3 threads(16, 16);
7      dim3 blocks((width + 15) / 16, (height + 15) / 16);
8      MandelbrotSet<<<blocks, threads>>>(image, startX, startY, dx, dy, width, height);
9
10     cudaGLUnmapBufferObject(buffer);
11     ...
12     // bind buffer to a texture and draw it
13 }
```

wissen.leben
WWU Münster

› Mandelbrotmenge: Geschwindigkeitsvergleich

CPU sequentiell | CPU OpenMP | GPU CUDA

Speedup: 25, 20, 15, 10, 5

100 | 1000 | 10000 | 65536

Maximale Iterationen

CPU: Intel i7 920 Quad Core, 2.67 GHz | GPU: NVIDIA GeForce 9800GX2

wissen.leben
WWU Münster

› Mandelbrotmenge: Geschwindigkeitsvergleich

CPU: Intel i7 920 Quad Core, 2.67 GHz | GPU: NVIDIA GeForce 9800GX2

wissen.leben
WWU Münster

› Mandelbrotmenge: Geschwindigkeitsvergleich

CPU: Intel i7 920 Quad Core, 2.67 GHz | GPU: NVIDIA GeForce 9800GX2

wissen.leben WWU Münster

› Mandelbrotmenge: Geschwindigkeitsvergleich

CPU sequentiell | CPU OpenMP | GPU CUDA

Speedup: 200, 160, 120, 80, 40

100 | 1000 | 10000 | 65536

Maximale Iterationen

CPU: Intel i7 920 Quad Core, 2.67 GHz | GPU: NVIDIA GeForce 9800GX2

wissen.leben WWU Münster

› Mandelbrotmenge: Geschwindigkeitsvergleich

ZIVHPC | GPU CUDA

Laufzeit in Sekunden: 20, 15, 10, 5

65536

Maximale Iterationen

ZIVHPC: 10 Knoten mit je 2 Quad Core AMD Opteron CPUs, 2.1 GHz | GPU: NVIDIA GeForce 9800GX2

wissen.leben
WWU Münster

› Kantendetektion mit dem Sobel Operator

- Ziel der Kantendetektion ist es in einem gegebenen Bild Kanten zu finden und zu markieren
- Die hier gezeigt Kantendetektion führt dazu eine Faltung mit dem Sobel Operator durch

wissen.leben WWU Münster

› Faltung mit dem Sobel Operator

- › Die Berechnung der Pixels des Ergebnisbildes G sind unabhängig

$$S_x := \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad S_y := \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

$$G_x(x,y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} S_x(i,j) \cdot F(x+i, y+j)$$

$$G_y(x,y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} S_y(i,j) \cdot F(x+i, y+j)$$

$$G(x,y) = |G_x(x,y)| + |G_y(x,y)|$$

wissen.leben WWU Münster

› CUDA Kernel: Sobel Operator

```
1  __global__ void Sobel(unsigned char* oImage, int imageWidth, float brightness) {
2       unsigned char* oRow = oImage + blockIdx.x * imageWidth;
3       int i = threadIdx.x;
4       int j = blockIdx.x;
5       unsigned char pix00 = tex2D( tex, (float) i-1, (float) j-1 );
6       unsigned char pix01 = tex2D( tex, (float) i+0, (float) j-1 );
7       unsigned char pix02 = tex2D( tex, (float) i+1, (float) j-1 );
8       ...
9
10      oRow[i] = ComputeSobel( pix00, pix01, pix02,
11                              pix10, pix11, pix12,
12                              pix20, pix21, pix22, brightness );
13 }
```

wissen.leben WWU Münster

› Host Code: CUDA Texture Unit

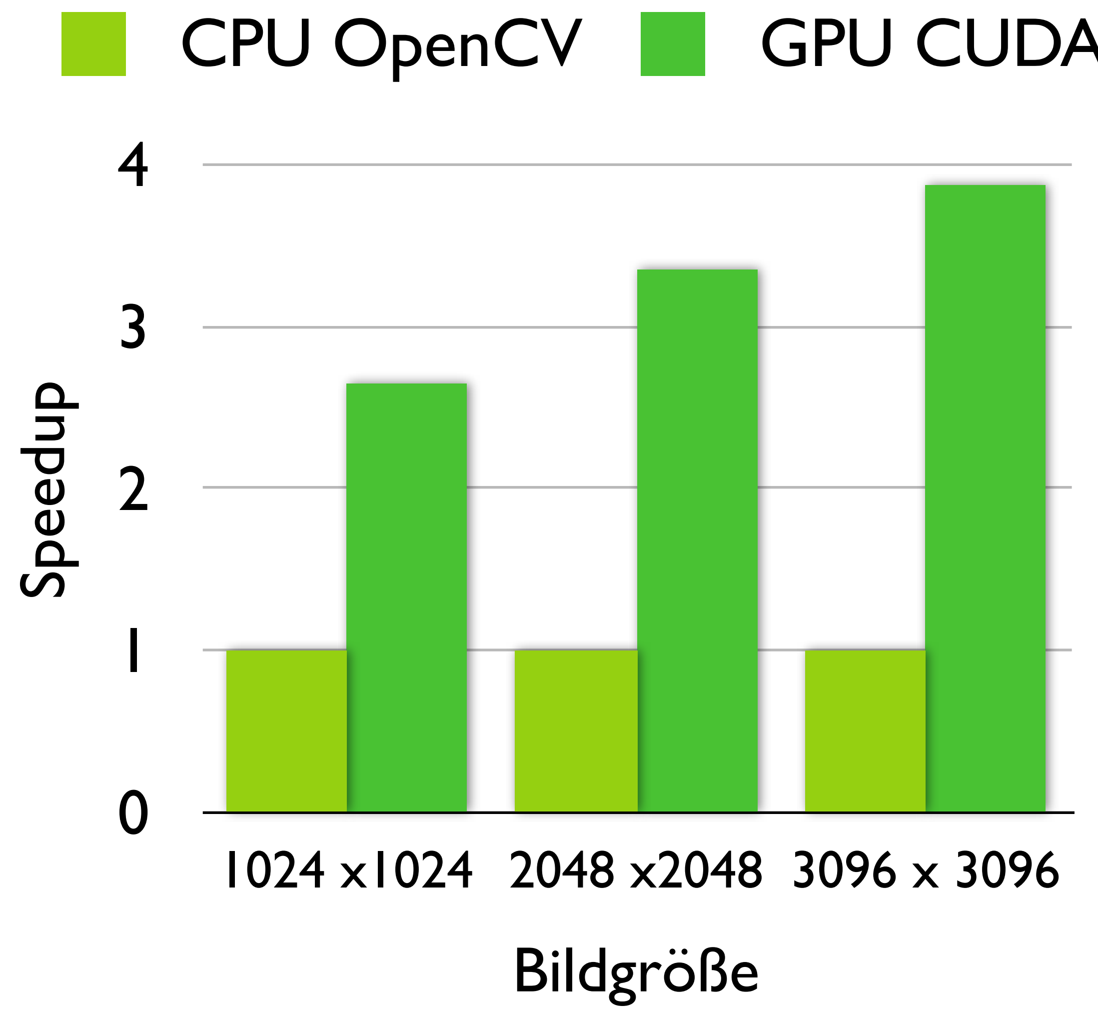
```
1  ...
2  desc = cudaCreateChannelDesc<unsigned char>();
3  cudaMallocArray(&array, &desc, imageWidth, imageHeight);
4  cudaMemcpyToArray( array, 0, 0, pixels,
5                     sizeof(unsigned char) * imageWidth * imageHeight,
6                     cudaMemcpyHostToDevice );
7  ...
8  cudaBindTextureToArray(tex, array);
9  Sobel<<<imageHeight, min(sobelWidth, 512)>>>( oImage, imageWidth, brightness );
10 cudaUnbindTexture(tex);
11 ...
```

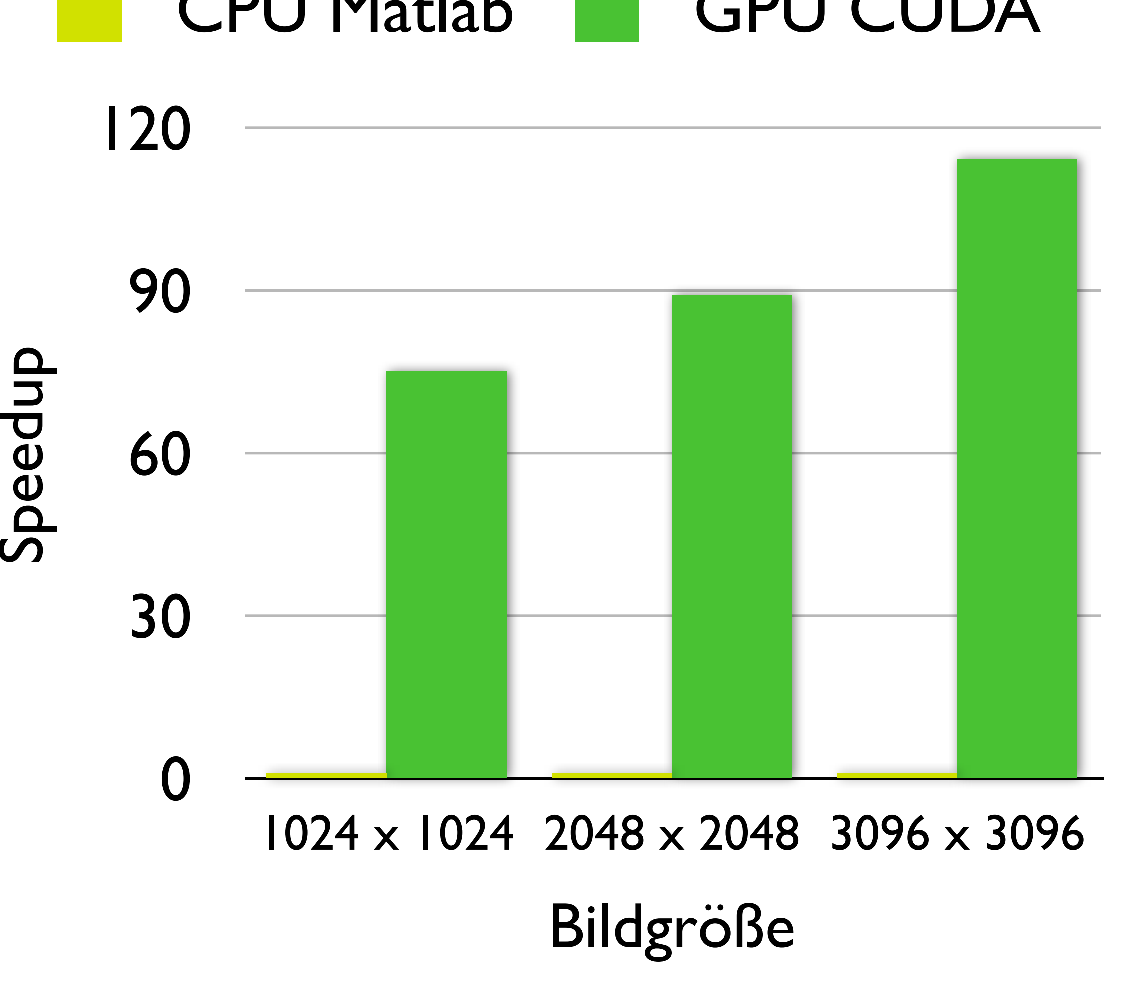
wissen.leben
WWU Münster

› Kantendetektion mit Canny: Geschwindigkeitsvergleich

Y. M. Luo; R. Duraiswami : Canny Edge Detection on NVIDIA CUDA

Details unter: http://terpconnect.umd.edu/~yluo1/canny.htm





wissen.leben
WWU Münster

› Programmierbeispiele und Implementierung

Viele Dank für Ihre Aufmerksamkeit

Gibt es Fragen?