03-MB-709.03

Echtzeitbildverarbeitung (6)

Prof. Dr. Udo Frese

C/C++ Optimierungen
Multi-core Parallelisierung
SIMD Parallelisierung

Was bisher geschah

- **Faltungsoperationen**
  - Lineare, translationsinvariante Abbildung
  - Ergebnispixel ist gewichtete Summe der Pixel in der Umgebung des Eingangspixels
  - Faltung mit einem Filter
    - *Filter **gespiegelt** auf Bild an der jeweiligen Position legen*
    - *übereinanderliegende Quellpixel/Filterkoeffizienten multiplizieren*
    - *Produkte aufaddieren und in Zielpixel schreiben*
  - Bilder glätten, Kontrast vergrößern, Kanten detektieren
- **Kantendetektion mit dem Sobel Filter (SobelX, SobelY)**
  - SobelX und SobelY geben als Ergebnis einen Vektor
  - Betrag: Kantenstärke
  - Richtung: Richtung der Kante (senkrecht zum Helleren)

1	0	-1
2	0	-2
1	0	-1

1	2	1
0	0	0
-1	-2	-1

C/C++ Optimierungen

C/C++ Optimierungen

Motivation

- **Meistens ist Entwicklungszeit wertvoller als Rechenzeit**
  - Computer sind sehr schnell (ca. 25000 mal schneller seit 25 Jahren)
  - Compiler optimieren selbsttätig (-O3)
  - Mehr Projekte scheitern, weil sie nicht fertig werden, als weil sie zu langsam rechnen
- **Bildverarbeitung verarbeitet enorme Datenmengen**
  - Bsp (Ballfangen): 2Kameras*25Bilder/s*2MPixel/Bild = 100M Pixel/s
  - Rechenzeit immer noch großes Problem
  - Trotzdem: Erst Funktionalität, dann Rechenzeit optimieren.
- **Folgende Optimierungen nur bei sehr wichtigen Teilroutinen**

C/C++ Optimierungen

Effizienz durch Tabellen (LUT)

- **für: komplizierte Rechnungen mit wenigen Variablen**
  - Beispiel: Farbklassifikation aus R, G, B
  - spezielle Funktionen, z.B. statistische Quantile (erf, chi^2)
  - historisch: Tabellenwerke für exp, log, sin, cos
- **Idee: Tabelle mit Ergebnis für jede Kombination der Variablen**
- **mehrere Ergebnisse zur selben Variablenkombination möglich**
- **gut, wenn Variablen schon diskret / diskretisiert sind**
- **Zusatzrechnungen in Tabelle integrieren**
  - Diskretisierung, z.B. Pixel auf ganze Koordinaten
  - Normalisierung, z.B. Winkel auf [0..360] Grad
  - Beschränkung des Ergebnisses, z.B. Helligkeit auf [0..255]
  - t.w. Adressberechnungen für z.B. Bildzugriff

C/C++ Optimierungen

Festkommaarithmetik

- **reelle Zahlen als ganzen Zähler mit festem Nenner darstellen**
- **oft 2^i als Nenner**
- **Beispiel: 0.5 → 128/256, nur 128 gespeichert**
- **vermeidet `float` oder `double` (nicht primärer Vorteil).**
- **vermeidet Konversionen `int-float/double`.**
- **Einsatz in Routinen, die eigentlich in `int` rechnen, aber zwischendrin eine Formel mit reellen Zahlen haben**
  - Pixelwerte (meist `unsigned char`)
  - Koordinaten (`int`)
- **ermöglicht direkten Tabellenzugriff.**

C/C++ Optimierungen

- **Frage an das Auditorium: Wie würde untenstehender Code in Festkomma-Arithmetik aussehen (zur Basis 1/256=1/2^8)?**

```
double scalar (double x0, double y0, double x1, double y1)
{
   return x0*x1+y0*y1;
}
```

C/C++ Optimierungen

- **Frage an das Auditorium: Wie würde untenstehender Code in Festkomma-Arithmetik aussehen (zur Basis $1/256=1/2^8$)?**

```
double scalar (double x0, double y0, double x1, double y1)
{
   return x0*x1+y0*y1;
}

int scalar (int x0, int y0, int x1, int y1)
{
   return (x0*x1+y0*y1)>>8;
}
```

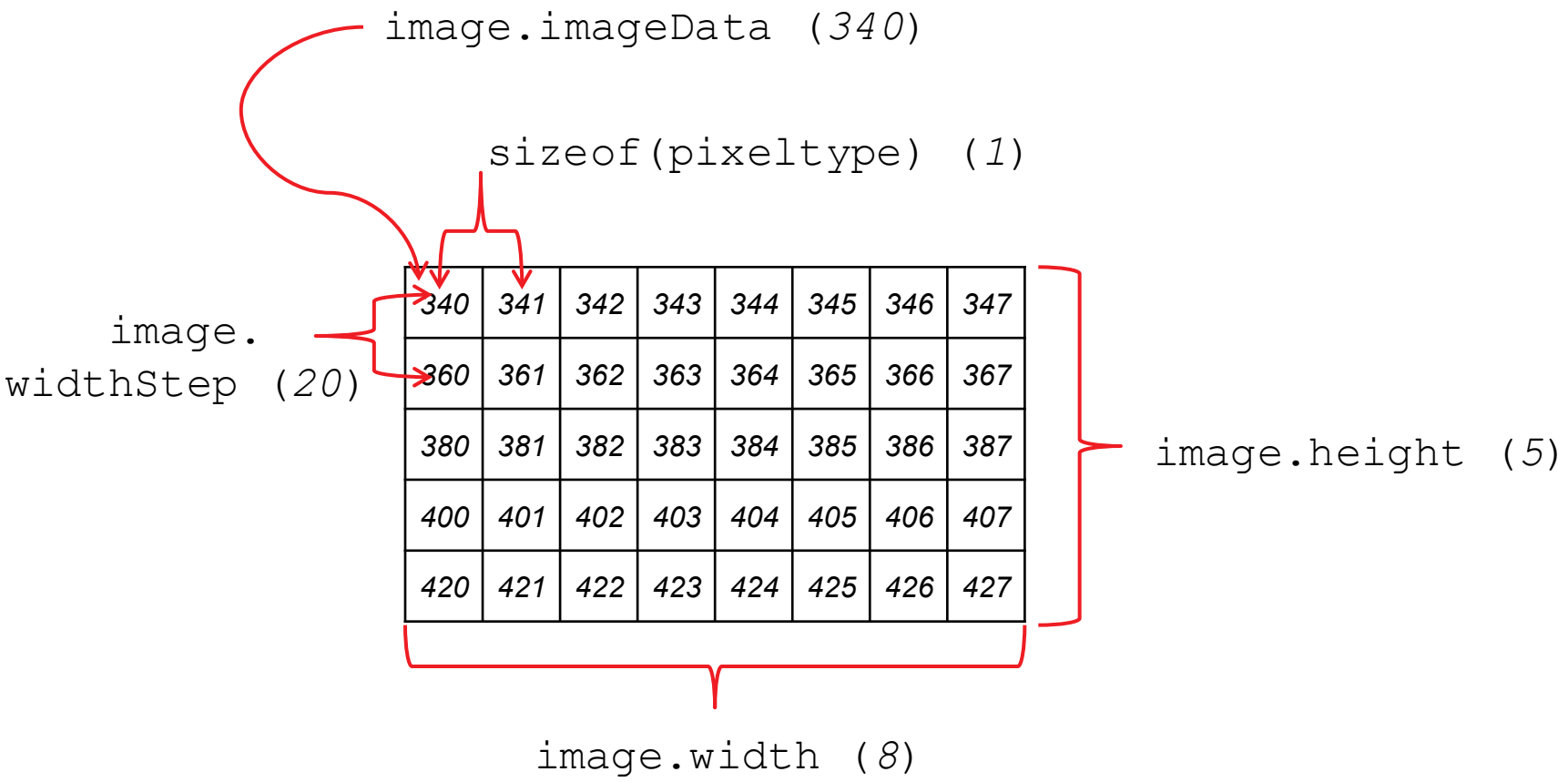
C/C++ Optimierungen

Festkommaarithmetik zur Basis 2^n

- **Konversion int nach Festkomma: i → (i<<n)**
- **Konversion Festkomma nach int: a → ((a+2^n/2)>>n)**
- **Addition: a+b → a+b**
- **Subtraktion: a-b → a-b**
- **Multiplikation mit int c: a*c→ a*c**
- **Multiplikation: a*b → ((a*b+2^n/2)>>n)**
  - +n/2 um zu runden
- **Division: a/b → ((a<<n+b/2)/b)**
  - +b/2 um zu runden
- **ACHTUNG: Immer (…) um << und >>**

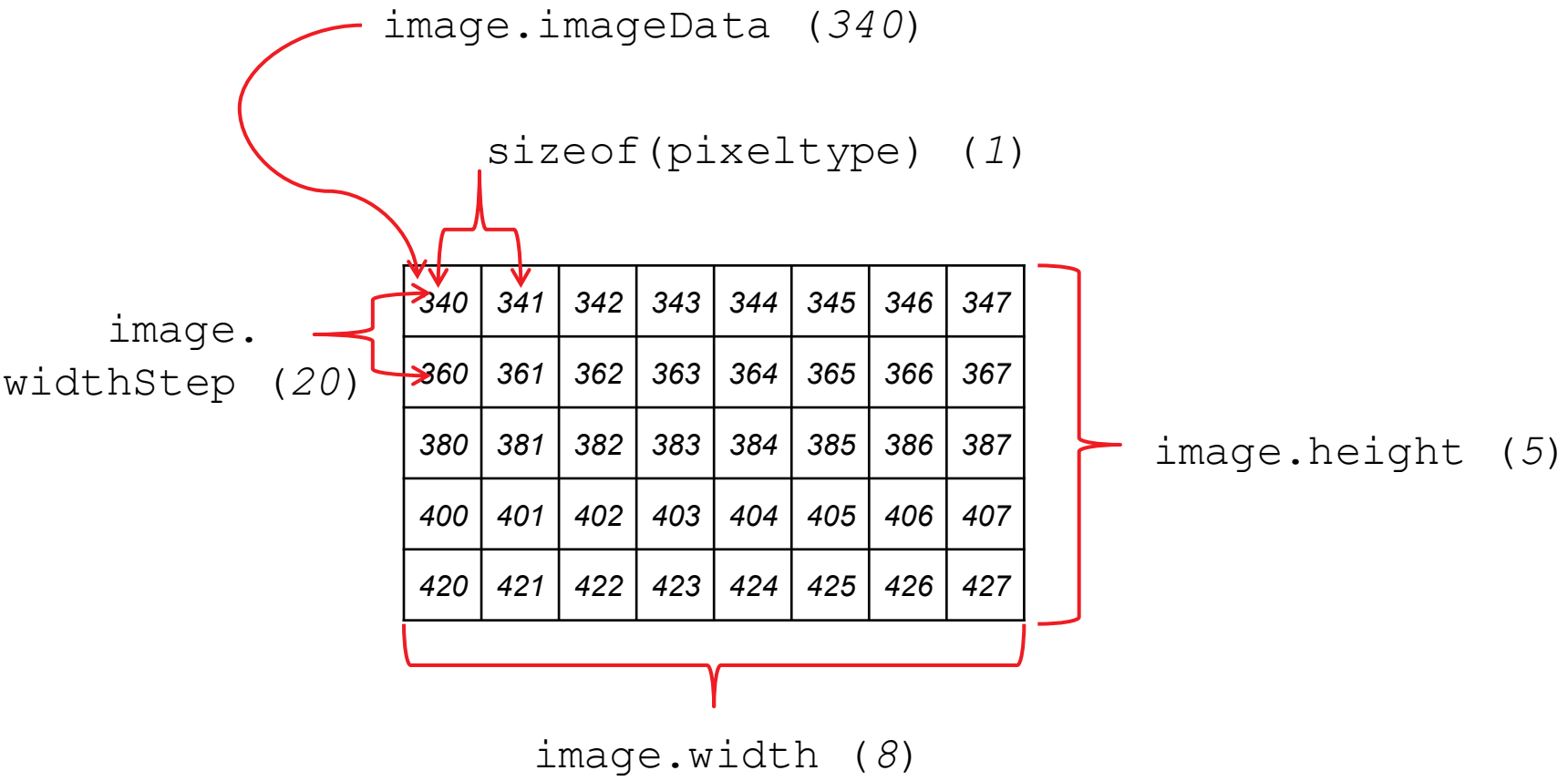
Aufbau eines Bildes im Speicher

- Zeiger in C++ verweisen auf eine Speicherstelle
- als Zahl durch Adresse beschrieben
- Pixel in einer Zeile liegen an aufeinanderfolgenden Adressen
- Zeilen folgen im Abstand von image.widthStep



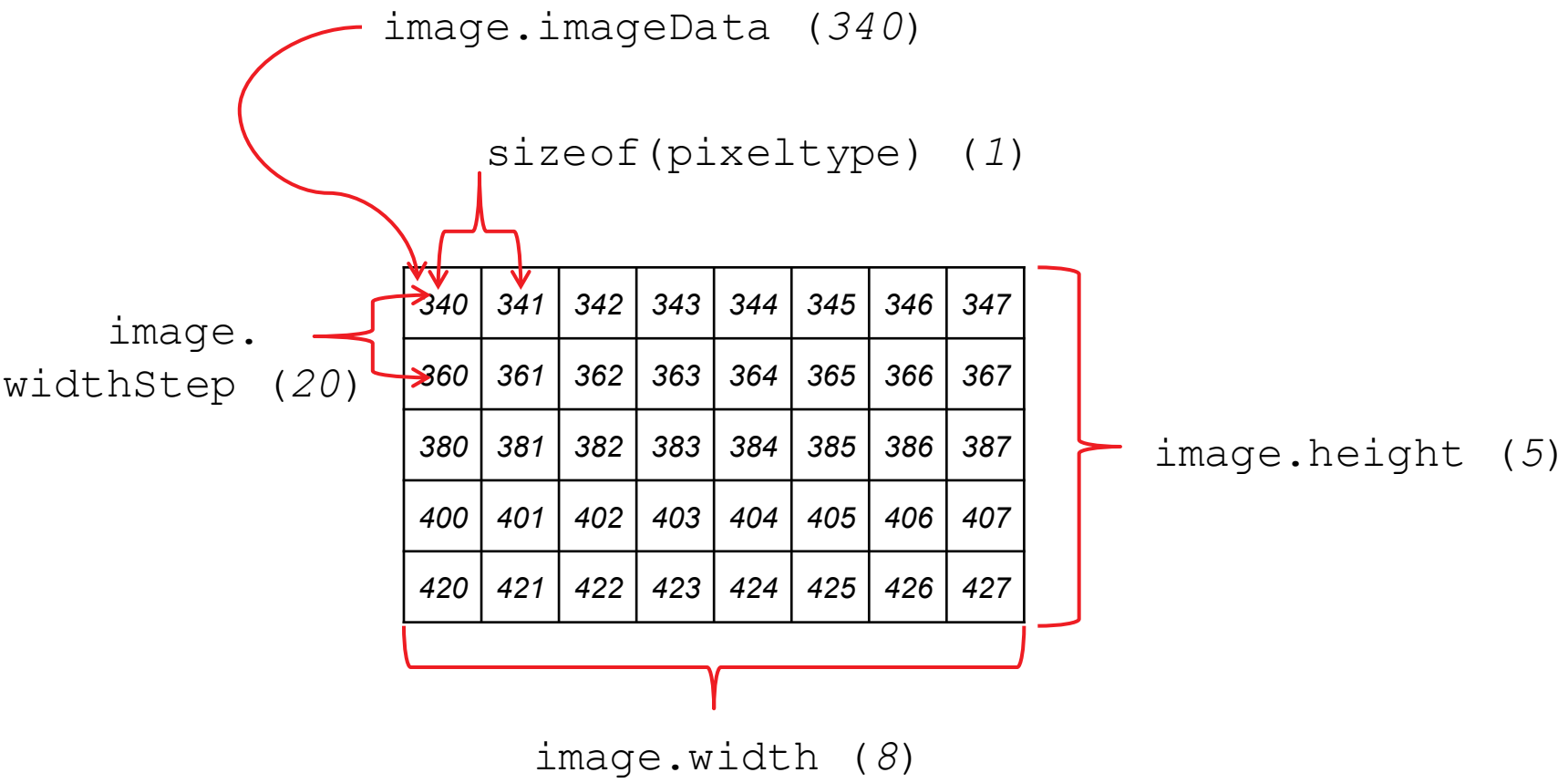
Aufbau eines Bildes im Speicher

- **Frage an das Auditorium: Warum trennt man zwischen `.width` und `.widthStep`?**



Aufbau eines Bildes im Speicher

- **Frage an das Auditorium: Warum trennt man zwischen `.width` und `.widthStep`?**
- **Um Ausschnitt eines Bildes als Bild betrachten zu können.**



C/C++ Optimierungen

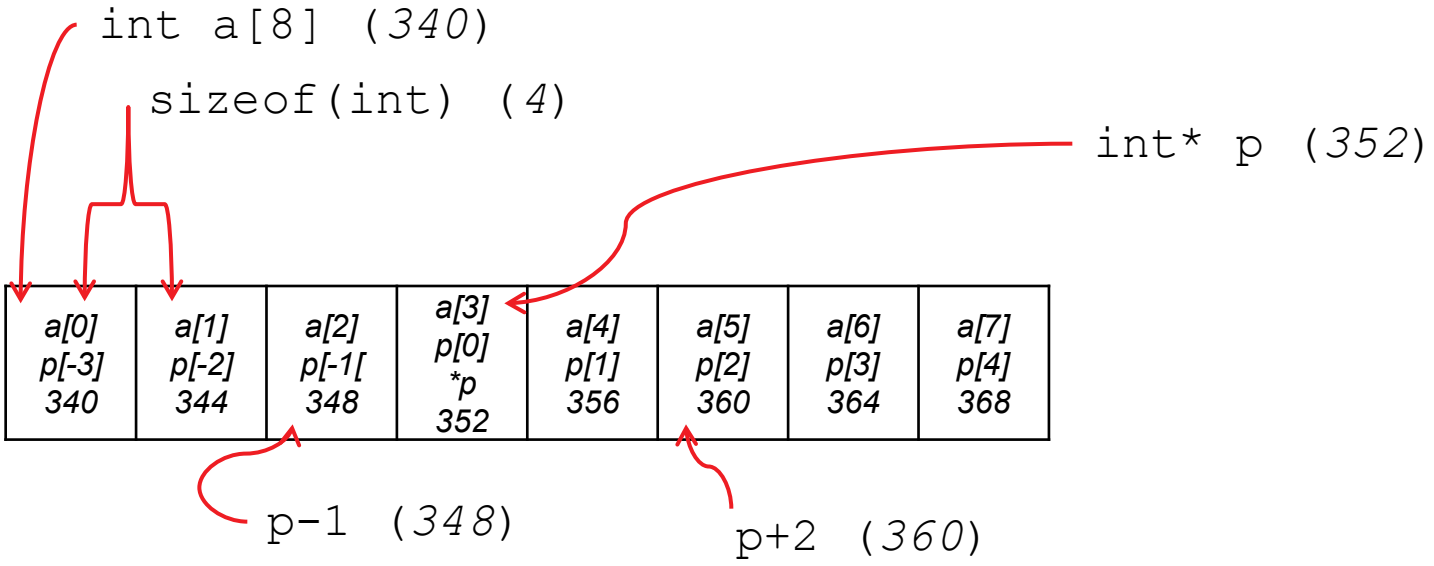
Zeigerarithmetik in C/C++

- **In C/C++ kann man mit Zeigern rechnen**
  - Vorteil: Schneller Zugriff in strukturierten Daten (Arrays, Bilder)
  - Nachteil: Möglichkeit, unkontrolliert irgendwas zu überschreiben
    - *crash, Sicherheitslücke: buffer overflow exploit*
- **Technisch: Rechnen mit der Adresse**

C/C++ Optimierungen

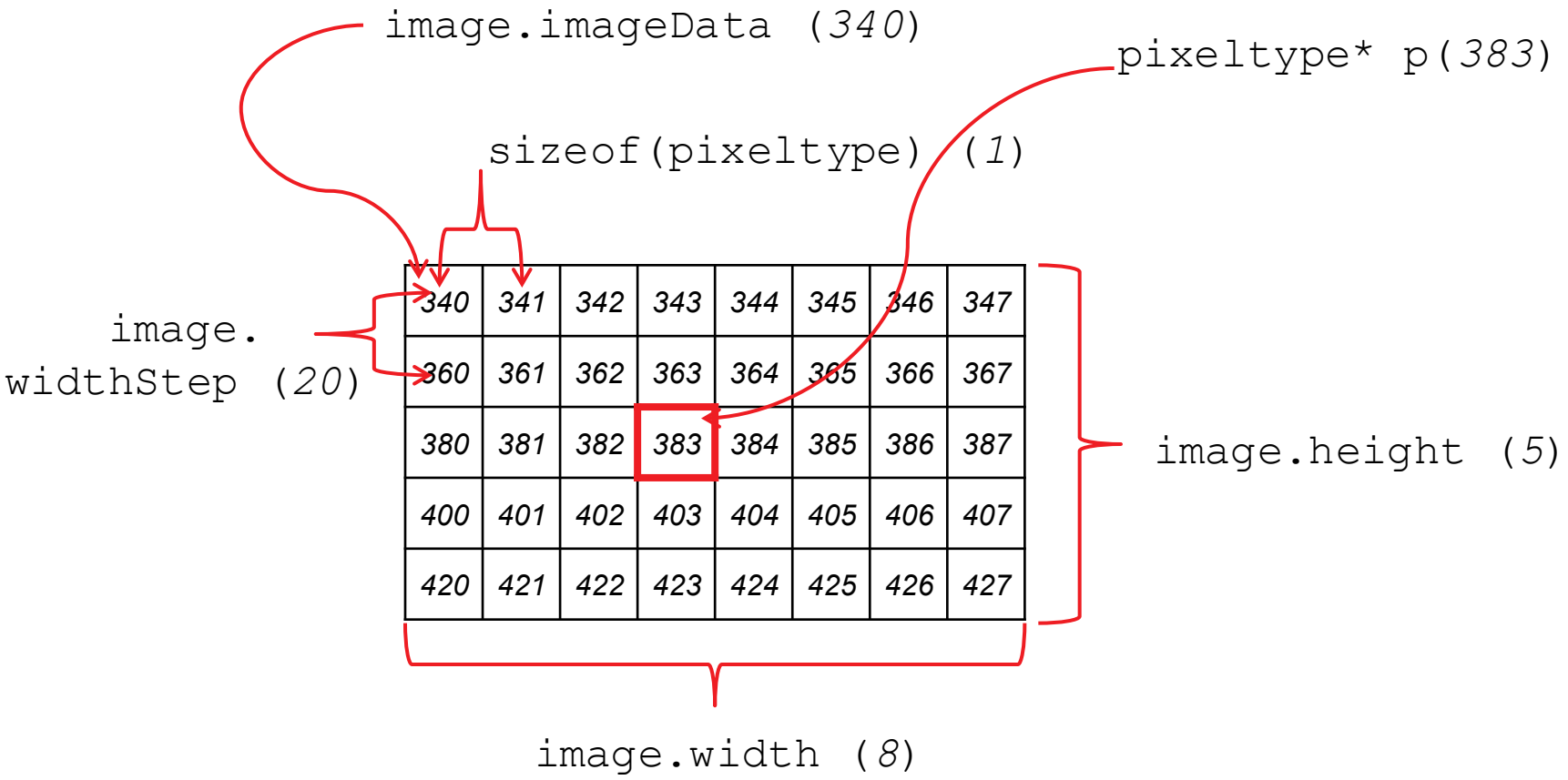
Zeigerarithmetik in C/C++

- **Zeigt `p` auf ein Element im Speicher**
  - `*p` das Element
  - `p+i` Zeiger auf `i`-te Element nach (`i`>0) oder vor (`i`<0) `p`
  - `p[i]` oder `*(p+i)` das `i`-te Element nach (`i`>0) oder vor (`i`<0) `p`
  - Elementgröße wird vom Compiler berücksichtigt



Aufbau eines Bildes im Speicher

- **Frage an das Auditorium: `p` zeigt auf einen Pixel im Bild. Wie greift man auf `p`'s 8 Nachbarn zu?**



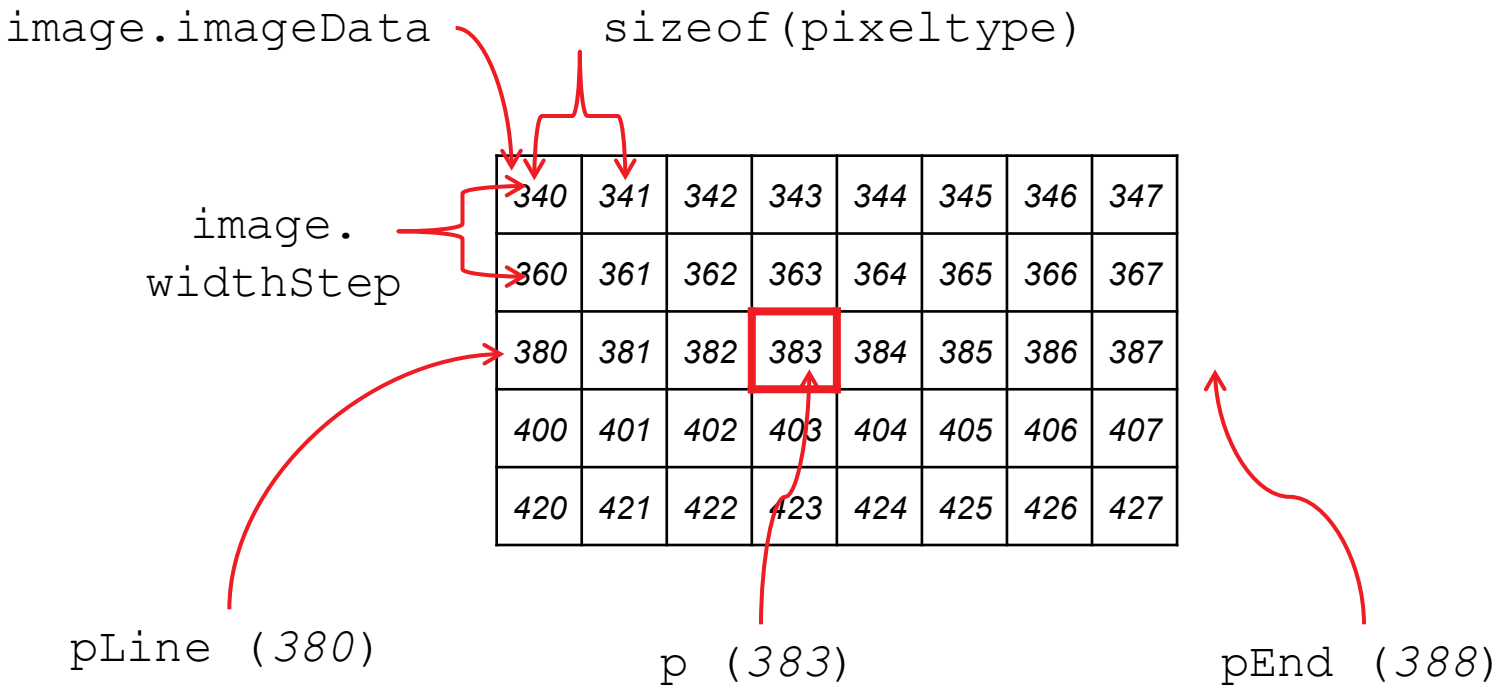
Aufbau eines Bildes im Speicher

- **Frage an das Auditorium: `p` zeigt auf einen Pixel im Bild. Wie greift man auf `p`'s 8 Nachbarn zu?**
- **Mit p[ ],**
  - jeweils 1 für X,
  - jeweils `ws = image.widthStep/sizeof(pixeltype)` für Y
    - *weil `widthStep` in Bytes nicht Pixeln ist*
    - *und Compiler mit `sizeof(pixeltype)` malnimmt*

p[-ws-1]	*p[-ws]*	*p[-ws+1]*
p[-1]	**p* *p[0]*	*p[1]*
p[+ws-1]	*p[+ws]*	*p[+ws+1]*

```
ws = image.widthStep/sizeof(pixeltype)
```

Durchlaufen eines Bildes im Speicher




```
void workOnImage (Image& img)
{
   pixel *p, *pEnd, *pLine = (pixel*) img.imageData;
   for (int y=0; y<img.height; y++) {
      for (p=pLine, pEnd = p+img.width; p<pEnd; p++) {
         *p = …; // Operate on pixel *p
      }
      pLine += dstImg->widthStep/sizeof(pixel);
   }
}
```

C/C++ Optimierungen

Direkte Implementierung einer Faltung

- 4 geschachtelte Schleifen
- Bildzugriff über Funktionen / Operatoren
- Randtest in innerster Schleife (äquivalent zu 0 außerhalb)

```
void convolve (Image& dstImg, Image& srcImg, DoubleImage& filter)
{
   for (y2=0; y2<dstImg.height; y2++)
      for (x2=0; x2<dstImg.width; x2++) {
         double sum = 0;
         for (yF=0; yF<filter.height; yF++)
            for (xF=0; xF<filter.width; xF++) {
               x = x2 - (xF - filter.width/2);
               y = y2 - (yF - filter.height/2);
               if (0<=xS && x<srcImg.width &&
                  0<=y && y<srcImg.height)
                  sum += filter(x2, y2) * srcImg(x, y);
            }
         dstImg(x2, y2) = sum + filter.offset;
      }
}
```

C/C++ Optimierungen

Direkte Implementierung einer Faltung

- ggf. Sonderbehandlung für Rand

```
void convolve (Image& dstImg, Image& srcImg, DoubleImage& filter)
{
   for (y2=0; y2<dstImg.height; y2++)
      for (x2=0; x2<dstImg.width; x2++) {
         double sum = 0; bool inside = true;
         for (yF=0; yF<filter.height; yF++)
            for (xF=0; xF<filter.width; xF++) {
               x = x2 - (xF - filter.width/2);
               y = y2 - (yF - filter.height/2);
               if (0<=x && x<srcImg.width &&
                  0<=y && y<srcImg.height)
                  sum += filter(x2, y2) * srcImg(x, y);
               else inside = false;
            }
         if (inside) dstImg(x2, y2) = sum + filter.offset;
         else dstImg (x2, y2) = marginResult;
      }
}
```

C/C++ Optimierungen

Frage an das Auditorium: Wie könnte man die Implementierung effizienter machen? Wie für einen speziellen Filter?

```
void convolve (Image& dstImg, Image& srcImg, DoubleImage& filter)
{
   for (y2=0; y2<dstImg.height; y2++)
      for (x2=0; x2<dstImg.width; x2++) {
         double sum = 0; bool inside = true;
         for (yF=0; yF<filter.height; yF++)
            for (xF=0; xF<filter.width; xF++) {
               x = x2 - (xF - filter.width/2);
               y = y2 - (yF - filter.height/2);
               if (0<=x && x<srcImg.width &&
                  0<=y && y<srcImg.height)
                  sum += filter(x2, y2) * srcImg(x, y);
               else inside = false;
            }
         if (inside) dstImg(x2, y2) = sum + filter.offset;
         else dstImg (x2, y2) = marginResult;
      }
}
```

C/C++ Optimierungen

Beschleunigung der Faltung durch

- **Allgemeine Faltung**
  - Zeiger statt Koordinaten
  - Rand in äußeren 2 Schleifen berücksichtigen ⇒ innen kein Test
  - rechnen mit `int`, Summe durch ggT teilen
- **Spezieller Filter**
  - inneren 2 Schleifen durch Term ersetzen (!)
  - 0 Koeffizienten auslassen
  - gleiche Koeffizienten zusammenfassen
  - Multiplikation / Division mit Zweierpotenzen durch <<, >> (autom.)
  - direkten Zugriff auf Nachbarpiel über p[…]
  - konstanten Speicheroffset von 1 für x ausnutzen

0	1/8	0
1/8	4/8	1/8
0	1/8	0

Optimierte Implementierung für 3*3 Glättungsfilter

```
void blur3x3 (Image& dstImg, Image& srcImg)
{
   int wss = srcImg->widthStep/sizeof(pixelSrc);
   int wsd = dstImg->widthStep/sizeof(pixelDst);
   pixelDst *p, *pEnd, *pLine = (pixelDst*) dstImg.imageData;
   pixelSrc *pSrc, *pSrcLine = (pixelSrc*) srcImg.imageData;
   for (p = pLine, pEnd = p+dstImg.width; p<pEnd; p++)
    *p = marginResult;
   pLine += wsd;
   pSrcLine += wss;
   for (int y2=1; y2<dstImg.height-1; y2++) {
      for (pSrc = pSrcLine, p=pLine, pEnd = p+dstImg.width;
           p<pEnd; p++, pSrc++) {
         int sum =              (int) pSrc[-wss] +
            (int) pSrc[-1] + ((int) pSrc[0]<<2) + (int) pSrc[1]
                               (int) pSrc[+wss];
         *p = (sum+4)>>3;
      }
    pLine[0] = pLine[dstImg.width-1] = marginResult;
    pLine += wsd;
    pSrcLine += wss;
  }
  for (p = pLine, pEnd = p+dstImg->width; p<pEnd; p++)
    *p = marginResult;
}
```

0	1/8	0
1/8	4/8	1/8
0	1/8	0

Multi-core Parallelisierung

Multi-core Parallelisierung

OpenMP

- **Fast alle heutigen Prozessoren haben mehrere Kerne**
  - Kern = Recheneinheit die eigenes Programm (Thread) ausführt
  - alle Kerne teilen den Speicher
  - Aufgaben auf Kerne verteilen, nicht Daten verteilen
- **Viele Algorithmen sind einfach parallelisierbar**
  - Z.B. Filter: Jeder Pixel unabhängig berechenbar
- **Herausforderung**
  - Wie programmiert man einfache Parallelisierung möglichst einfach?

Multi-core Parallelisierung

OpenMP

- **serielles Programm schreiben**
- **mit Sonderanweisungen `#pragma` dem Compiler sagen, was parallel laufen kann**
- **"mein Programm befehligt ein Team von Kernen (Threads)"**
  - *openMP teilt Teamaufgaben auf Teammitglieder auf*
  - *openMP koordiniert die Mitglieder*
  - einfache Lösung bei einfachen Fällen
- **www.openmp.org**
- **Seung-Jai Min, OpenMP Tutorial**
  https://engineering.purdue.edu/~eigenman/ECE563/Handouts/ECE563-OpenMP.pdf

Parallele for-Schleifen

- **`#pragma omp parallel`**
  - Definiert eine parallele Sektion
  - von allen Threads gemeinsam durchlaufen
  - darin deklarierte Variablen hat jeder Thread für sich
  - lokale Variablen in aufgerufenen Funktionen hat jeder Thread für sich
  - außerhalb deklarierte Variablen sind gemeinsam für alle Threads
- **#pragma omp for**
  - Definiert eine parallele Schleife
  - Ausführung der einzelnen Durchläufe ist unabhängig
  - openMP teilt Durchläufe auf Threads auf
  - Kann in Routine stehen, die von paralleler Sektion aus aufgerufen wird
  - ACHTUNG: Nicht `#pragma omp parallel for`

```
void add (vector<int>& a, vector<int>& b)
{
   #pragma omp parallel
   {
      #pragma omp for
      for (int i=0; i<a.size(); i++)
         a[i] += b[i];
   }
}
```

Parallele for-Schleifen

- **Frage an das Auditorium: Wo ist hier der Fehler?**

```
void runningSum (vector<int>& a)
{
   #pragma omp parallel
   {
      #pragma omp for
      for (int i=0; i<a.size(); i++)
         if (i>0) a[i] = a[i]+a[i-1];
   }
}
```

Parallele for-Schleifen

- Frage an das Auditorium: Wo ist hier der Fehler?
- Die Schleifendurchläufe sind nicht unabhängig
- In der Tat ist Laufsummenberechnung schwer zu parallelisieren

```
void runningSum (vector<int>& a)
{
   #pragma omp parallel
   {
      #pragma omp for
      for (int i=0; i<a.size(); i++)
         if (i>0) a[i] = a[i]+a[i-1];
   }
}
```

Parallele for-Schleifen

- Frage an das Auditorium: Wo ist hier der Fehler?

```
int sum (vector<int>& a)
{
   #pragma omp parallel
   {
      #pragma omp for
      int sum=0;
      for (int i=0; i<a.size(); i++)
         sum += a[i];
   }
   return sum;
}
```

Parallele for-Schleifen

- Frage an das Auditorium: Wo ist hier der Fehler?
- Jeder Thread hat seine eigene Teilsumme in sum und es wird nur die Summe des Hauptthreads zurückgeliefert.
- Was ist hier falsch?

```
int sum (vector<int>& a)
{
   int sum=0;
   #pragma omp parallel
   {
      #pragma omp for
      for (int i=0; i<a.size(); i++)
         sum += a[i];
   }
   return sum;
}
```

Parallele for-Schleifen

- **Was ist hier falsch?**
- **Die Threads erhöhen gemeinsam sum, wenn lesen und schreiben sich kreuzt, gehen Erhöhungen verloren.**
- **#pragma omp critical**
  - Kritische Sektion
  - Nur ein Thread gleichzeitig drin
  - Hier zu viel Overhead,
  - Sinnvoll bei mehr Rechnungen pro Eintritt in die Kritische Sektion

```
int sum (vector<int>& a)
{
   int sum=0;
   #pragma omp parallel
   {
      #pragma omp for
      for (int i=0; i<a.size(); i++)
         #pragma omp critical
         { sum += a[i]; }
   }
   return sum;
}
```

Parallele for-Schleifen

- **Was ist hier falsch?**
- **Die Threads erhöhen gemeinsam sum, wenn lesen und schreiben sich kreuzt, gehen Erhöhungen verloren.**
- **`reduction (op:name)`**
  - Gibt jedem Thread eine eigene Variable `name`
  - Zum Schluss werden Ergebnisse mit `op` verknüpft

```
int sum (vector<int>& a)
{
   int sum=0;
   #pragma omp parallel
   {
      #pragma omp for reduction (+:sum)
      for (int i=0; i<a.size(); i++)
         sum += a[i];
   }
   return sum;
}
```

Parallelisierbare Implementierung für 3*3 Glättungsfilter

- Adressberechnung in jeder Zeile neu
- Randzeilen über if / else
- ⇒Parallelisierung über Zeilen möglich

```
void blur3x3 (Image& dstImg, Image& srcImg)
{
   int wss = srcImg->widthStep/sizeof(pixelSrc);
   int wsd = dstImg->widthStep/sizeof(pixelDst);
   for (int y2=0; y2<dstImg.height; y2++) {
      pixelDst *p, *pEnd, *pLine = (pixelDst*) dstImg.imageData + y2*wsd;
      pixelSrc *pSrc, *pSrcLine = (pixelSrc*) srcImg.imageData + y2*wss;
      if (y2>0 && y2<dstImg.height-1) {
         for (pSrc = pSrcLine, p=pLine, pEnd = p+dstImg.width;
              p<pEnd; p++, pSrc++) {
            int sum =          (int) pSrc[-wss] +
               (int) pSrc[-1] + ((int) pSrc[0]<<2) + (int) pSrc[1]
                                (int) pSrc[+wss];
            *p = (sum+4)>>3;
         }
          pLine[0] = pLine[dstImg.width-1] = marginResult;
      }
      else for (p = pLine, pEnd = p+dstImg->width; p<pEnd; p++)
             *p = marginResult;
   }
}
```

OpenMP Implementierung für 3*3 Glättungsfilter

```
void blur3x3 (Image& dstImg, Image& srcImg)
{
   #pragma omp parallel
   {
      int wss = srcImg->widthStep/sizeof(pixelSrc);
      int wsd = dstImg->widthStep/sizeof(pixelDst);
      #pragma omp for
      for (int y2=0; y2<dstImg.height; y2++) {
         pixelDst *p, *pEnd, *pLine =(pixelDst*)dstImg.imageData+ y2*wsd;
         pixelSrc *pSrc, *pSrcLine = (pixelSrc*)srcImg.imageData+ y2*wss;
         if (y2>0 && y2<dstImg.height-1) {
            for (pSrc = pSrcLine, p=pLine, pEnd = p+dstImg.width;
                 p<pEnd; p++, pSrc++) {
               int sum =                (int) pSrc[-wss] +
                  (int) pSrc[-1] + ((int) pSrc[0]<<2) + (int) pSrc[1]
                                        (int) pSrc[+wss];
               *p = (sum+4)>>3;
            }
             pLine[0] = pLine[dstImg.width-1] = marginResult;
         }
         else for (p = pLine, pEnd = p+dstImg->width; p<pEnd; p++)
                *p = marginResult;
      }
   }
}
```

SIMD Parallelisierung

SIMD Parallelisierung

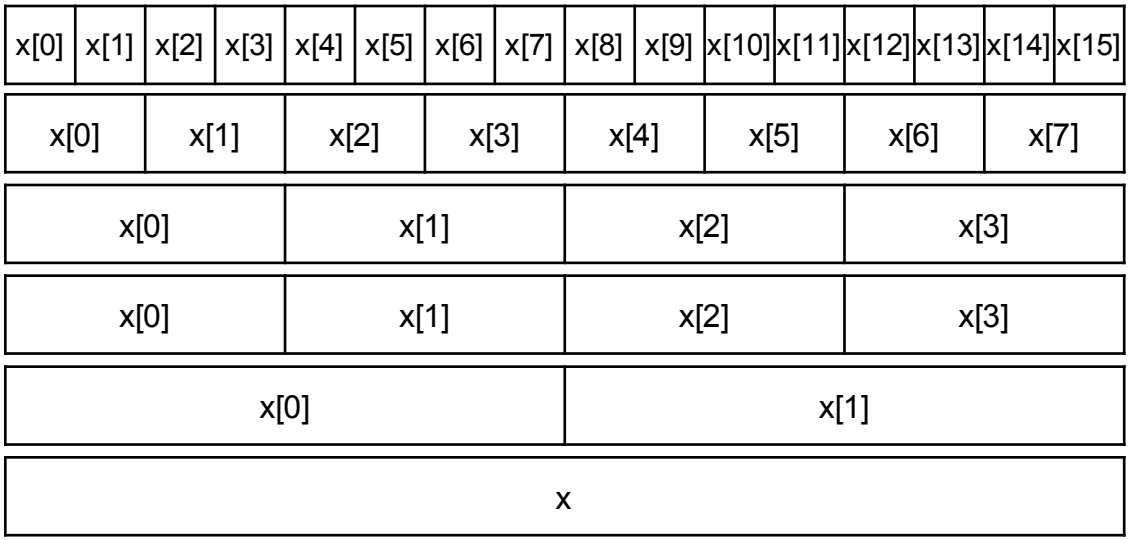
Motivation

- **OpenMP Parallelisierung**
  - verschiedene Threads führen dasselbe Programm aus
  - aber nicht dieselbe Anweisung
  - Bedingungen fallen unterschiedlich aus
  - größere Blöcke von Arbeit parallelisieren
  - jeder Kern hat eigene Recheneinheit und eigene Kontrolleinheit
- **Single Instruction Multiple Data**
  - Befehl wendet die selbe Operation auf mehrere Werte gleichzeitig an
  - exakt dieselbe Anweisung
  - Elementare Operationen parallelisieren
  - eine Kontrolleinheit, mehrere Recheneinheiten
- **Intels MMX, SSE-SSE4 Befehlssatz**
  - Intel Intrinsics Reference, #312482-002US,
    http://cache-www.intel.com/cd/00/00/34/76/347603_347603.pdf

SIMD Parallelisierung

SIMD Datentypen

- **Datentyp für 128 Bits**
  - SSE
  - andere Systeme ähnlich
- **Interpretiert als**
  - (unsigned) char x[16]
  - (unsigned) short x[8]
  - (unsigned) int x[4]
  - float x[4]
  - double x[2]
  - 128 Bits
- **Direkt vom Prozessor unterstützt**
  - 128 Bit Register
  - Spezialbefehle



SIMD Parallelisierung

SIMD Datentypen

- **Datentypen in C/C++ (einbinden mit `#include <emmintrin.h>`)**
  - (unsigned) char [16] , (unsigned) short [8] , (unsigned) int [4] → `__m128i`
  - float [4] → `__m128`
  - double[2] → `__m128d`
- **Variablendeklaration**
  - `__m128i srcPixel, dstPixel;`
  - automatisch *ausgerichtet*, d.h. Adresse durch 16 teilbar
  - vom Compiler automatisch in Register verschoben
- **Uminterpretation von Arraydatentypen**
  - `short a[1000]; __m128i* p = (__m128i*) &(a[7]);`
  - `p` Zeiger auf Block von 8 short-Zahlen, (`a[7], a[8], .., a[14]`)
  - typischerweise nicht *ausgerichtet (*kompliziertes Problem*)*
  - Ausnahme: Zeilenanfänge von Bildern *ausgerichtet*
  - Sofern Bilder mit durch 16 teilbarer Breite und keine Unterbilder!

SIMD Parallelisierung

SIMD Operationen

- **Verarbeiten i.a. ein oder zwei SIMD Daten**
- **Im Prozessor als Befehl ⇒ schnell**
- **In C/C++ als "intrinsic" Funktion**
- **Syntax: `_mm_`*operation*`_`*typ*`(...)`**
- ***operation*: Kürzel für die Verknüpfung**
  - z.B. `add, sub, mul, and, or, …`
- ***typ*: Interpretation der SIMD Daten als**
  - (unsigned) char x[16] → epi8 (epu8), "*extended packed integer 8 bit*"
  - (unsigned) short x[8] → epi16 (epu16)
  - (unsigned) int x[4] → epi32 (epu32)
  - float x[4] → ps, "*packed single precision*"
  - double x[2] → pd, "*packed double precision*"
  - 128 Bits → si, "*single integer*"

SIMD Parallelisierung

Zuweisungen

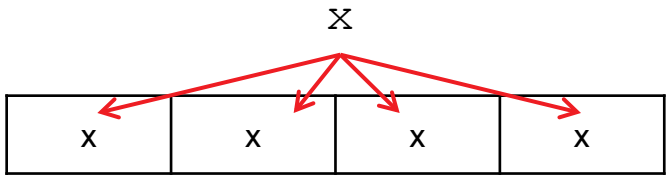
- **Ausgerichtete Variablen per = zuweisen**
- **Unausgerichtete Variablen laden ("*load unaligned*")**
  - `__m128i _mm_loadu_si128 (const __m128i* p)`
  - `__m128 _mm_loadu_ps (const float* p)`
  - `__m128d _mm_loadu_pd (const double* p)`
  - wie *p, aber nicht ausgerichtet
  - etwas langsamer als ausgerichtet
- **Unausgerichtete Variablen speichern ("*store unaligned*")**
  - `_mm_storeu_si128 (__m128i* p, __m128i x)`
  - `_mm_storeu_ps (float* p, __m128 x)`
  - `_mm_storeu_pd (double* p, __m128d x)`
  - wie *p = x, aber nicht ausgerichtet
  - merklich langsamer als ausgerichtet

SIMD Parallelisierung

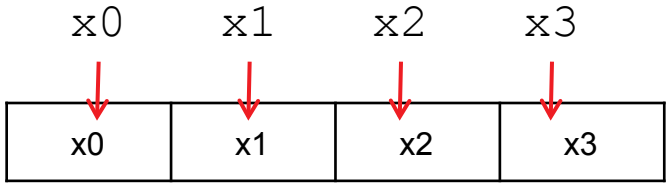
Zuweisungen

- **Null**
  - `__m128i _m_setzero_si128 ()`
  - `__m128 _m_setzero_ps ()`
  - `__m128d _m_setzero_pd ()`

- **Vektor gleicher Werte**
  - `__m128i _m_set1_typ (typ x)`
  - `__m128 _m_set1_ps (float x)`
  - `__m128d _m_set1_pd (double x)`



- **Vektor einzelner Werte**
  - `__m128i _m_set_typ (…)`
  - `__m128 _m_set_ps (float x0, float x1, float x2, float x3)`
  - `__m128d _m_set_pd (double x0, double x1)`



SIMD Parallelisierung

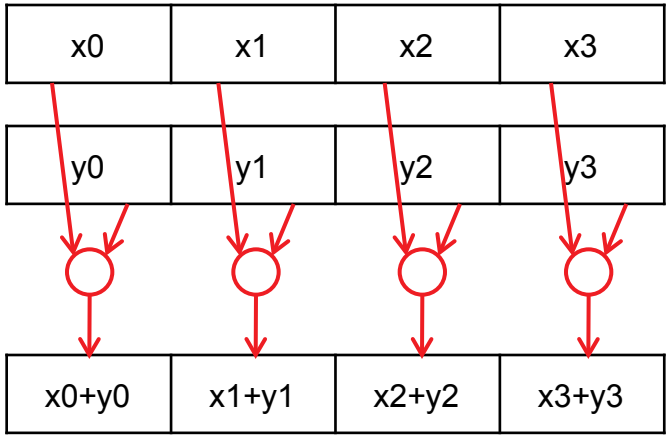
- **Beispiel: Füllen eines Bildes mit einer Farbe**

```
void fill (IplImage* img, unsigned char color)
{
   unsigned char *p, *pEnd, *pLine = img->imageData;
   __m128i color16 = _mm_set1_epu8 (color);
   for (int y=0; y<img->height; y++) {
      for (p=pLine, pEnd=p+img->width; p<pEnd; p+=16)
         *((__m128i*)p) = color16;
// or _mm_storeu_si128 (p, color16) if unaligned
      pLine += img->widthStep/sizeof(unsigned char);
   }
}
```

SIMD Parallelisierung

Arithmetik (Fließkomma)

- **Verknüpfe korrespondiere Einträge zweier Operanden ("vertikal")**
- **Addition, Subtraktion, Multiplikation, Division, Min, Max**
  - `__m128 _mm_add_ps (__m128 x, __m128 y)`
  - `__m128d _mm_add_pd (__m128d x, __m128d y)`
  - `_mm_sub_p?, _mm_mul_p?, _mm_div_p?, _mm_min_p?, _mm_max_p?`
- **Kehrwert, Kehrwert der Wurzel,Wurzel**
  - `__m128 _mm_rcp_ps (__m128 x)`
  - `__m128d _mm_rcp_pd (__m128d x)`
  - `__m128 _mm_rsqrt_ps (__m128 x)`
  - `__m128d _mm_rsqrt_pd (__m128d x)`
- **Skalarprodukt!**
  - `_mm_dp_ps, _mm_dp_pd`
  - spezielle Optionen



SIMD Parallelisierung

Arithmetik (integer)

- **Addition, Subtraktion (Überlauf/Sättigung)**
  - Überlauf (`unsigned char`): 240+20 = 4
  - Sättigung (`unsigned char`): 240+20 = 255
  - `_mm_add_?, _mm_sub_?`
  - `_mm_adds_?, _mm_subs_?`
- **Multiplikation (Überlauf) ("Multiply low")**
  - `__m128i _mm_mullo_epi? (__m128i a, __m128i b)`
  - nur `epi16`, `epi32` und äquivalent `epu16`, `epu32`
- **Weitere spezielle Multiplikationsbefehle**
- **Maximum, Minimum**
  - `_mm_max_?, _mm_min_?`
- **Betrag**
  - `_mm_abs_epi?`

SIMD Parallelisierung

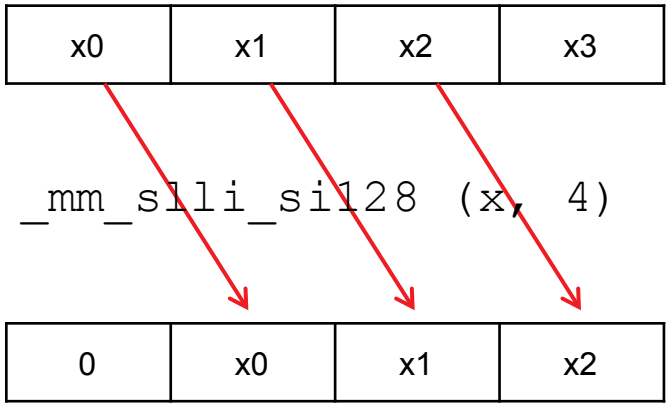
Logik (integer)

- **Bitweises und, oder, und-nicht, exclusiv-oder**
  - `_mm_and_si128, _mm_or_si128, _mm_andnot_si128, _mm_xor_si128`
- **Schieben für Rechnen in den Grenzen der einzelnen Werte**
- **Linkschieben ("shift left logical by immediate")**
  - `__m128i _mm_slli_epi? (__m128i x, int nBits)`
  - nur `epi?`, auch bei `epu?`
  - $*2^{nBits}$
- **Rechtsschieben ohne Vorz. ("shift right logical by immediate)**
  - `__m128i _mm_srli_epi? (__m128i x, int nBits)`
  - eigentlich `epu?`, aber `epi?` definiert
  - $/2^{nBits}$
- **Rechtsschieben mit Vorz. ("shift right arithmetic by immediate)**
  - `__m128i _mm_srai_epi? (__m128i x, int nBits)`
  - $/2^{nBits}$

SIMD Parallelisierung

Logistik

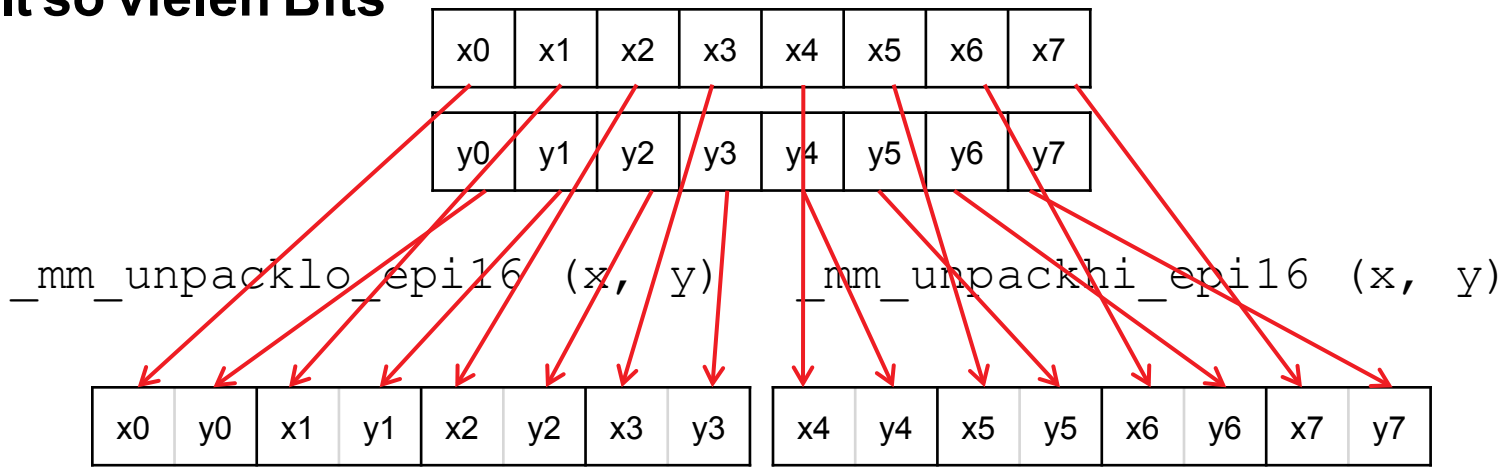
- **Allgemeines Permutieren der Werte ("shuffle")**
  - `_mm_shuffle_? (…)`
  - mächtig aber kompliziert zu benutzen
- **Schieben über die Grenzen der Einzelwerte hinweg**
  - `__m128i _mm_slli_si128 (__m128i x, int nBytes)`
  - left: byte[i+nBytes] = byte[i]
  - `__m128i _mm_srli_si128 (__m128i x, int nBytes)`
  - right: byte[i-nBytes] = byte[i]
  - Schiebt um nBytes Bytes = nBytes*8 Bits



SIMD Parallelisierung

Konvertierung

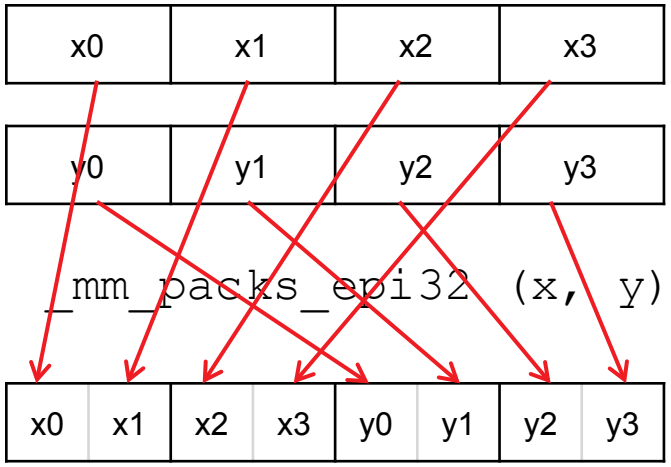
- klein nach gross: epu8→epu16, epu16→epu32
- `_mm_unpacklo_? (x, y), __mm_unpack_hi_? (x, y)`
- ?: epi8, epi16 (äquivalent zu epu8, epu16)
- abwechselnd Werte von `x` und `y` der unteren bzw. oberen Hälfte
- mit `y=0`: konvertiere untere/obere Hälfte von `x` von ? Bits nach doppelt so vielen Bits



SIMD Parallelisierung

Konvertierung

- groß nach klein: ep?32→ep?16, ep?16→ep?8,
- `_mm_packs_?  (x, y)`
- konvertiert `x` und `y` von `?` Bits auf `?/2` Bits und hängt `y` hinter `x`
- bei Konversion werden zu große/kleine Werte beschränkt (geclipt)
- `_mm_packus_epi?` statt eigentlich `_mm_packs_epu?`



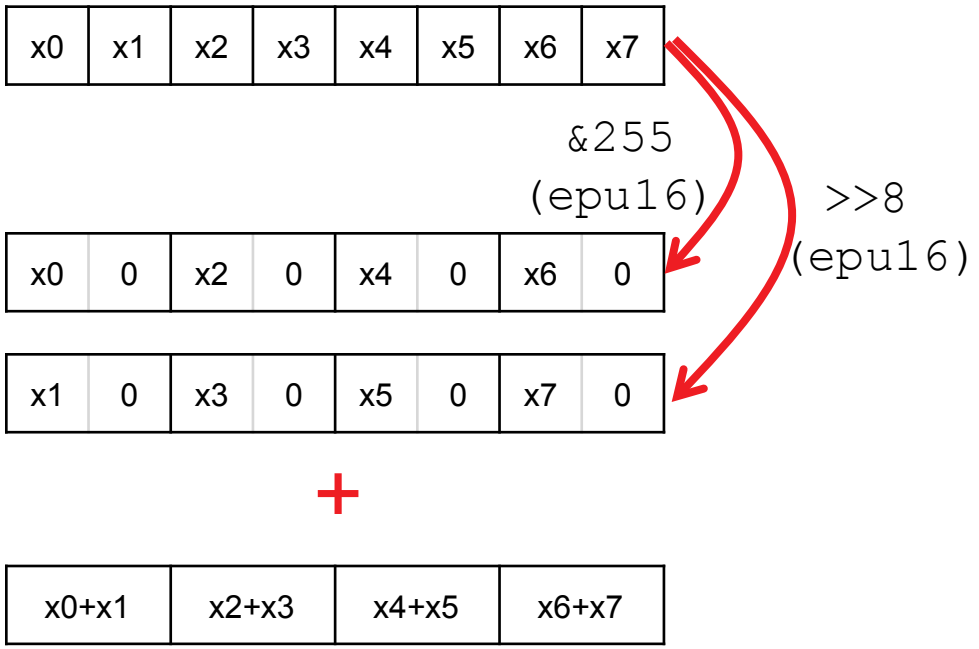
SIMD Parallelisierung

- **Beispiel: Halbieren der Auflösung eines Bildes**
- **Pixel im Ausgabebild ist Mittelwert von 2*2 Pixeln im Eingabebild (Summe/4)**

```
void scaleByHalf (IplImage* dstImg, IplImage* srcImg)
{
   unsigned char *p, *pEnd, *pLine = dstImg->imageData;
   unsigned char *pSrc, *pSrcLine = srcImg->imageData;
   int wss = srcImg->widthStep/sizeof(unsigned char);
   for (int y=0; y<dstImg->height; y++) {
      for (p=pLine, pSrc=pSrcLine, pEnd=p+dstImg->width;
           p<pEnd; p++, pSrc+=2) {
            *p = ((int) pSrc[0  ]+(int) pSrc[    1]
                 +(int) pSrc[wss]+(int) pSrc[wss+1]+2)>>2;
      }
      pLine   += dstImg->widthStep/sizeof(unsigned char);
      pSrcLine+= 2*wss;
   }
}
```

SIMD Parallelisierung

- **Beispiel: Halbieren der Auflösung eines Bildes mit SIMD**
- **16 Pixel laden**
- **2 Probleme mit `((int) p[0]+(int) p[1]+…)`**
  - Summe muss epu16 sein
  - Pixel horizontal addieren
- **Zwischenergebnisse vertikal addieren**
- **2 nach rechts schieben**
- **Konvertieren auf epu8**
- **Speichern von nur 8 Pixeln**
  - `_mm_storel_epi64 (p, x)`
  - Spezialbefehl!



Beispiel: Halbieren der Auflösung eines Bildes mit SIMD

```
void scaleByHalf (IplImage* dstImg, IplImage* srcImg)
{
   unsigned char *p, *pEnd, *pLine = dstImg->imageData;
   unsigned char *pSrc, *pSrcLine = srcImg->imageData;
   int wss = srcImg->widthStep/sizeof(unsigned char);
   __m128i c255_epu16 = _mm_set1_epi16 (255);
   __m128i c2_epu16 = _mm_set1_epi16 (2);
   for (int y=0; y<dstImg->height; y++) {
      for (p=pLine, pSrc=pSrcLine, pEnd=p+dstImg->width;
           p<pEnd; p+=8, pSrc+=16) {
         __m128i src0 = *((__m128i*)(pSrc+0));
         __m128i sum0= _mm_add_epi16(_mm_and_si128 (src0, c255_epu16),
                                     _mm_srli_epi16 (src0, 8));
         __m128i src1 = *((__m128i*)(pSrc+wss));
         __m128i sum1= _mm_add_epi16(_mm_and_si128 (src1, c255_epu16),
                                     _mm_srli_epi16 (src1, 8));
         __m128i sum = _mm_add_epi16(sum0, sum1),;
         sum = _mm_packs_epi16 (
            _mm_srli_epi16 (_mm_add_epi16(sum, c2_epu16), 2)
            _mm_setzero_si128());
         _mm_storel_epi64 ((__m128i*) p, sum);
      }
      pLine   += dstImg->widthStep/sizeof(unsigned char);
      pSrcLine+= 2*wss;
   }
}
```

SIMD Parallelisierung

Warnungen!

- **SIMD Programmierung ist mühsam**
  - Puzzlespiel mit den vorhandenen Befehlen und Bitgrößen
  - Adressen und Längen müssen Vielfaches von 16 sein
  - Schrittweise C Routine und äquivalente SIMD Routine koentwickeln und Gleichheit prüfen
- **Gründe für geringe Performance**
  - Overhead für Parallelisierung zu groß
  - Speicherzugriff dominiert Rechenzeit
- **Manchmal lohnt sich SIMD Programmierung**
  - einfache Aufgabenstellung
  - Fließkommarechnungen
  - Wichtige Routine
  - weitere Spezialbefehle in Intel Intrinsics Reference

Zusammenfassung

- **Optimierungen in C/C++**
  - Tabellen
  - Zeiger statt Koordinaten zum Durchlaufen von Bildern
  - Zugriff auf Nachbarn über relative Adresse
  - Festkommaarithmetik
- **Multi-core Parallelisierung**
  - OpenMP: Schleifen parallelisieren durch Compilerhinweis `#pragma omp for`
  - Schleifendurchläufe müssen (im wesentlichen) unabhängig sein
- **SIMD Parallelisierung**
  - Die selbe Operation auf 4/8/16/… Werte anwenden
  - Daten müssen blockweise verarbeitet werden, Rechenreihenfolge fest
  - `if` nur sehr eingeschränkt
  - Vektordatentyp `__m128i`, `__m128`, `__m128d`
  - Verknüfungen `_mm_`*operation_typ* (..,..) werden in Befehl umgesetzt
  - technisch mit vielen Komplikationen