New code must be parallel!
otherwise a computer from 2016 is as slow as a computer from 2000…

But it can be easy!
a few basic principles, plus some knowledges of tools…
Introduction

• Modern computers have high-performance parallel processors
  • multicore CPU with vector units & pipelining
  • massively multicore GPU

• How to use them efficiently in practice?
Introduction

• Not just for high-end servers but also for *everyday programming tasks*
  • laptops, desktops, mobile devices…

• Sometimes you can easily improve running times *from minutes to seconds*
Key challenges

• Getting data from memory to processor
  • memory lookups are expensive
  • efficient use of memory hierarchy, caches

• Exploiting all parallel processing capacity
  • multicore, vector operations, pipelining …
  • necessary: lots of independent operations
An example
Image processing

• 2D array, 16000 x 16000 values, 32-bit ints
  • approx. 1 GB of data

• Median filter:
  • new value = median of pixel and its 4 neighbours
static void median(const array_t x, array_t y) {
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < n; ++i) {
            int ia = (i + n - 1) % n; int ib = (i + 1) % n;
            int ja = (j + n - 1) % n; int jb = (j + 1) % n;
            y[i][j] = median(x[i][j], x[ia][j], x[ib][j],
                             x[i][ja], x[i][jb]);
        }
    }
}
Baseline

static void median(const array_t x, array_t y) {
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < n; ++i) {
            int ia = (i + n - 1) % n; int ib = (i + 1) % n;
            int ja = (j + n - 1) % n; int jb = (j + 1) % n;
            y[i][j] = median(x[i][j], x[ia][j], x[ib][j],
                             x[i][ja], x[i][jb]);
        }
    }
}
Sanity checking

• Classroom computers: 3.3 GHz CPU, 4 cores, hyperthreading

• We are using > 800 clock cycles per pixel
  • median of 5 elements, should not be that hard?

• We are only using 1 thread on 1 core
Know the tools

```c
static void median(const array_t x, array_t y) {
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < n; ++i) {
            int ia = (i + n - 1) % n; int ib = (i + 1) % n;
            int ja = (j + n - 1) % n; int jb = (j + 1) % n;
            y[i][j] = median(x[i][j], x[ia][j], x[ib][j],
                             x[i][ja], x[i][jb]);
        }
    }
}
```

g++-4.8 -march=native -O3

running time: 25 s
Understand memory hierarchy

```c
static void median(const array_t x, array_t y) {
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < n; ++i) {
            int ia = (i + n - 1) % n; int ib = (i + 1) % n;
            int ja = (j + n - 1) % n; int jb = (j + 1) % n;
            y[i][j] = median(x[i][j], x[ia][j], x[ib][j],
                                x[i][ja], x[i][jb]);
        }
    }
}
```
static void median(const array_t x, array_t y) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            int ia = (i + n - 1) % n; int ib = (i + 1) % n;
            int ja = (j + n - 1) % n; int jb = (j + 1) % n;
            y[i][j] = median(x[i][j], x[ia][j], x[ib][j],
                              x[i][ja], x[i][jb]);
        }
    }
}
static void median(const array_t x, array_t y) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            int ia = (i + n - 1) % n; int ib = (i + 1) % n;
            int ja = (j + n - 1) % n; int jb = (j + 1) % n;
            y[i][j] = median(x[i][j], x[ia][j], x[ib][j],
                              x[i][ja], x[i][jb]);
        }
    }
}

running time: 9 s
Exploit parallel processing units

```c
static void median(const array_t x, array_t y) {
    #pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            int ia = (i + n - 1) % n; int ib = (i + 1) % n;
            int ja = (j + n - 1) % n; int jb = (j + 1) % n;
            y[i][j] = median(x[i][j], x[ia][j], x[ib][j],
                             x[i][ja], x[i][jb]);
        }
    }
}
```

g++-4.8 -fopenmp

running time: 2 s
## Running times

<table>
<thead>
<tr>
<th></th>
<th>Baseline</th>
<th>Compiler</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>68 s</td>
<td>25 s</td>
<td>9 s</td>
</tr>
<tr>
<td>Parallel</td>
<td>12 s</td>
<td>5 s</td>
<td>2 s</td>
</tr>
</tbody>
</table>
It can be this easy!

- Significant improvements in running times
  - from over a minute to a few seconds
- No algorithmic changes needed this time
- Memory layout: change array indexing
- Parallelisation: just add one #pragma
Are we done now?

- We are using \( \approx 20 \) clock cycles per pixel
  - sounds reasonable, but...
- Reading & writing memory \( \approx 1 \) GB/second
  - not a bottleneck yet, "memcpy" achieves > 18 GB/second
  - can we make the "median" function faster?
Better algorithms?

static int median(int v1, int v2, int v3, int v4, int v5) {
    int a[] = {v1, v2, v3, v4, v5};
    std::nth_element(a+0, a+2, a+5);
    return a[2];
}
Better algorithms?

static int median(int v1, int v2, int v3, int v4, int v5) {
    int a[] = {v1, v2, v3, v4, v5};
    for (int i = 0; i < 4; ++i) {
        int b = 0;
        for (int j = 0; j < 5; ++j) {
            b += (a[j] < a[i] || (a[i] == a[j] && i < j));
        }
        if (b == 2) {
            return a[i];
        }
    }
    return a[4];
}

“item with exactly 2 smaller items”
Better algorithms?

```java
static int median(int v1, int v2, int v3, int v4, int v5) {
    int a[] = {v1, v2, v3, v4, v5};
    for (int i = 0; i < 4; ++i) {
        int b = 0;
        for (int j = 0; j < 5; ++j) {
            b += (a[j] < a[i] || (a[i] == a[j] && i < j));
        }
        if (b == 2) {
            return a[i];
        }
    }
    return a[4];
}
```

Wait, what, $O(n^2)$ time??
Better algorithms?

- Implement a better “median” function:
  \[ \approx 0.6 \text{ s in total} \]
  \[ \approx 7 \text{ clock cycles per pixel} \]
  \[ \approx 4 \text{ GB/s} \]

- Are we happy now?
Know when to stop!

- Median filtering:
  \( \approx 0.6 \text{ s in total} \)

- Just copying data with "memcpy":
  \( \approx 0.1 \text{ s } \) (even after warm-up)
# Running times

<table>
<thead>
<tr>
<th></th>
<th>Baseline</th>
<th>Compiler</th>
<th>Memory</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>68 s</td>
<td>25 s</td>
<td>9 s</td>
<td>3 s</td>
</tr>
<tr>
<td>Parallel</td>
<td>12 s</td>
<td>5 s</td>
<td>2 s</td>
<td>0.6 s</td>
</tr>
</tbody>
</table>
What about GPUs?

cudaHostGetDevicePointer((void**)&outputGPU, outputCPU, 0);
cudaMalloc((void**)&inputGPU, size * sizeof(int));
cudaMemcpy(inputGPU, inputCPU, size * sizeof(int), cudaMemcpyHostToDevice);

dim3 dimBlock(64, 1);
dim3 dimGrid((width + dimBlock.x - 1) / dimBlock.x, (height + dimBlock.y - 1) / dimBlock.y);

medianKernel<<<dimGrid, dimBlock>>>(
        outputGPU, inputGPU, width, height, size
);

cudaFree(inputGPU);

(some boring details omitted...)
__global__ void medianKernel(int* output, const int* input, 
                           const int width, const int height, const int size) 
{
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    if (x >= width || y >= height) return;

    int p0 = x + width * y;
    int p1 = p0 - 1;    int p2 = p0 + 1;
    int p3 = p0 - width; int p4 = p0 + width;

    if (x == 0)          p1 += width;
    if (x == width - 1)  p2 -= width;
    if (y == 0)          p3 += size;
    if (y == height - 1) p4 -= size;

    int a0 = input[p0]; int a1 = input[p1]; int a2 = input[p2];
    int a3 = input[p3]; int a4 = input[p4];

    int b0 = min(a0, a1); int b1 = max(a0, a1); int b2 = min(a2, a3);
    int b3 = max(a2, a3); int c0 = min(b0, b2); int c2 = max(b0, b2);
    int c1 = min(b1, b3); int d1 = min(c1, c2); int d2 = max(c1, c2);
    int e4 = max(c0, a4); int f2 = min(d2, e4); int g2 = max(d1, f2);

    output[p0] = g2;
}
__global__ void medianKernel(int* output, const int* input,
                           const int width, const int height, const int size)
{
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    if (x >= width || y >= height) return;

    int p0 = x + width * y;
    int p1 = p0 - 1;    int p2 = p0 + 1;
    int p3 = p0 - width; int p4 = p0 + width;

    if (x == 0)       p1 += width;
    if (x == width - 1) p2 -= width;
    if (y == 0)       p3 += size;
    if (y == height - 1) p4 -= size;

    int a0 = input[p0]; int a1 = input[p1]; int a2 = input[p2];
    int a3 = input[p3]; int a4 = input[p4];

    int b0 = min(a0, a1); int b1 = max(a0, a1); int b2 = min(a2, a3);
    int b3 = max(a2, a3); int c0 = min(b0, b2); int c2 = max(b0, b2);
    int c1 = min(b1, b3); int d1 = min(c1, c2); int d2 = max(c1, c2);
    int e4 = max(c0, a4); int f2 = min(d2, e4); int g2 = max(d1, f2);

    output[p0] = g2;
}

running time: 0.3 s
Know when to stop!

- Median filtering with GPU:
  \(\approx 0.3\, \text{s in total}\)

- Just moving data to GPU and back:
  \(\approx 0.3\, \text{s}\)
About this course
Course overview

• Practical hands-on course, no exam
• Non-trivial algorithmic problems
• Everything happens on a single machine
  • no networking, no distributed computing
• Only wall-clock time matters
Only wall-clock time matters

• How many seconds does it take for this machine to solve this problem?

• Parallelism not a goal in itself, just one way to get more performance

• Benchmark everything!
  Do not assume, try it out and see yourself!
Assignments

• Plenty of tasks to choose from each week

• “Recommended path”:  
  • 10 pt each week, total 60 pt  
  • grading: 30 pt = 1/5, 50 pt = 5/5

• Contest: extra points for fastest solutions!
Workload

• 5 credits in 6 weeks ≈ 22 working hours per week
  • more than a half-time job!

• Lecture + exercises only 6 hours per week
  • you are expected to spend lots of time programming on your own
Tools

• C or C++ with OpenMP and vector extensions
  • e.g. Intel TBB is not that different

• C or C++ with CUDA
  • e.g. OpenCL is not that different
  • fairly easy to convert between CUDA and OpenCL once you know the basic principles
Tools

- Linux + usual Unix development tools
  - classroom: Maari-A

- Git + GitHub: github.com/ICS-E4020-2016

- Slack chat: aalttoppc.slack.com
  - you can be there right now!
Why parallelism?
Why parallelism?

Gordon E. Moore (1965): “Cramming more components onto integrated circuits”, *Electronics Magazine*

reprinted in *Proc. IEEE*, vol. 86, issue 1, Jan 1998
<table>
<thead>
<tr>
<th>Year</th>
<th>Transistors</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1975</td>
<td>3 000</td>
<td>6502</td>
</tr>
<tr>
<td>1979</td>
<td>30 000</td>
<td>8088</td>
</tr>
<tr>
<td>1985</td>
<td>300 000</td>
<td>386</td>
</tr>
<tr>
<td>1989</td>
<td>1 000 000</td>
<td>486</td>
</tr>
<tr>
<td>1995</td>
<td>6 000 000</td>
<td>Pentium Pro</td>
</tr>
<tr>
<td>2000</td>
<td>40 000 000</td>
<td>Pentium 4</td>
</tr>
<tr>
<td>year</td>
<td>Transistors</td>
<td>Clock speed</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>1975</td>
<td>3 000</td>
<td>1 000 000</td>
</tr>
<tr>
<td>1979</td>
<td>30 000</td>
<td>5 000 000</td>
</tr>
<tr>
<td>1985</td>
<td>300 000</td>
<td>20 000 000</td>
</tr>
<tr>
<td>1989</td>
<td>1 000 000</td>
<td>20 000 000</td>
</tr>
<tr>
<td>1995</td>
<td>6 000 000</td>
<td>200 000 000</td>
</tr>
<tr>
<td>2000</td>
<td>40 000 000</td>
<td>2 000 000 000</td>
</tr>
<tr>
<td>Year</td>
<td>Clock Cycles</td>
<td>Processor</td>
</tr>
<tr>
<td>------</td>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>1980</td>
<td>100</td>
<td>8087</td>
</tr>
<tr>
<td>1987</td>
<td>50</td>
<td>387</td>
</tr>
<tr>
<td>1993</td>
<td>3</td>
<td>Pentium</td>
</tr>
</tbody>
</table>

How many clock cycles does it take to do a floating-point multiplication (FMUL)?
Progress!

• Increasing:  
  clock cycles / second

• Decreasing:  
  clock cycles / operation

• Increasing rapidly:  
  operations / second
<table>
<thead>
<tr>
<th>Year</th>
<th>Transistors</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1975</td>
<td>3,000</td>
<td>6502</td>
</tr>
<tr>
<td>1979</td>
<td>30,000</td>
<td>8088</td>
</tr>
<tr>
<td>1985</td>
<td>300,000</td>
<td>386</td>
</tr>
<tr>
<td>1989</td>
<td>1,000,000</td>
<td>486</td>
</tr>
<tr>
<td>1995</td>
<td>6,000,000</td>
<td>Pentium Pro</td>
</tr>
<tr>
<td>2000</td>
<td>40,000,000</td>
<td>Pentium 4</td>
</tr>
<tr>
<td>2005</td>
<td>100,000,000</td>
<td>2-core Pentium D</td>
</tr>
<tr>
<td>2008</td>
<td>700,000,000</td>
<td>8-core Nahelem</td>
</tr>
<tr>
<td>2014</td>
<td>6,000,000,000</td>
<td>18-core Haswell</td>
</tr>
<tr>
<td>Year</td>
<td>Transistors</td>
<td>Clock speed</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>1975</td>
<td>3,000</td>
<td>1,000,000</td>
</tr>
<tr>
<td>1979</td>
<td>30,000</td>
<td>5,000,000</td>
</tr>
<tr>
<td>1985</td>
<td>300,000</td>
<td>20,000,000</td>
</tr>
<tr>
<td>1989</td>
<td>1,000,000</td>
<td>20,000,000</td>
</tr>
<tr>
<td>1995</td>
<td>6,000,000</td>
<td>200,000,000</td>
</tr>
<tr>
<td>2000</td>
<td>40,000,000</td>
<td>2,000,000,000</td>
</tr>
<tr>
<td>2005</td>
<td>100,000,000</td>
<td>3,000,000,000</td>
</tr>
<tr>
<td>2008</td>
<td>700,000,000</td>
<td>3,000,000,000</td>
</tr>
<tr>
<td>2014</td>
<td>6,000,000,000</td>
<td>2,000,000,000</td>
</tr>
</tbody>
</table>
How many clock cycles does it take to do a floating-point multiplication (FMUL)?
Progress???

• Not increasing:
  clock cycles / second

• Not decreasing:
  clock cycles / operation

• Not increasing:
  operations / second ???
Latency vs. throughput

- **Latency**: time to perform operation from start to finish
- **Throughput**: how many operations are completed per time unit
  - in the long run
Latency vs. throughput

• MSc degrees at Aalto:
  • *latency*: 2 years
  • *throughput*: 1663 degrees / year

• Note that Aalto is massively parallel
  • a sequential university would have a throughput of 0.5 degrees / year …
Dependent operations

\[ a_1 \times= a_0; \]
\[ a_2 \times= a_1; \]
\[ a_3 \times= a_2; \]
\[ a_4 \times= a_3; \]

Inherently sequential

Bottleneck: \textit{latency}

Independent operations

\[ b_1 \times= a_1; \]
\[ b_2 \times= a_2; \]
\[ b_3 \times= a_3; \]
\[ b_4 \times= a_4; \]

Opportunities for parallelism

Bottleneck: \textit{throughput}
New kind of progress

• Difficult: designing CPUs with faster multiplication units
  • latency not improving

• “Easy”: designing CPUs with a large number of parallel multiplication units
  • throughput improving
New kind of progress

• 1993 (Pentium):
  • 0.3 dependent multiplications / cycle
  • 0.5 independent multiplications / cycle

• 2014 (12-core Haswell):
  • 0.2 dependent multiplications / cycle
  • 100 independent multiplications / cycle
New kind of progress

• Not increasing:  
  dependent operations / second

• Increasing rapidly:  
  independent operations / second

• *All new performance comes from parallelism*
CPUs and GPUs
CPUs and GPUs used to be very different

- **CPU**: “central processing unit”, “processor”
  - *general-purpose* processor
  - ran *sequential* code, one instruction at time

- **GPU**: “graphics processing unit”
  - only *special primitives* for 3D graphics
  - inherently *parallel* (“do this for all points”)

Convergence: CPUs ≈ GPUs

• **CPU:**
  • lots of old code written for sequential CPUs
  • but new performance comes from parallelism
  • engineers tried to introduce features so that old code would benefit from parallelism
  • limited success, you really *need new code* designed for parallel machines
Convergence:
CPUs ≈ GPUs

• GPU:
  • more and more special primitives added to support more complicated 3D graphics
  • introduction of *programmable shaders*
  • towards unified design: from special-purpose primitives to *parallel general-purpose computers*
Convergence: CPUs $\approx$ GPUs

- CPUs and GPUs similar nowadays:
  - *general-purpose processors*: read/write memory, do arithmetics, branch, loop, etc.
  - *massively parallel processors*: hundreds of multiplications in progress at the same time
  - *new code needed*: programmers have to take parallelism into account (*but it can be easy!*
Convergence: CPUs ≈ GPUs

- Also some differences: programming interfaces, tradeoffs in internal structure ...

- Good reasons: different target applications

- Historical reasons: all GPU code is fairly new, CPUs try to keep old code happy
What kind of parallelism is there?

... and how to exploit it?
Bit-level parallelism

```c
bool a[64];
bool b[64];
for (int i = 0; i < 64; ++i) {
    a[i] = a[i] || b[i];
}

uint64_t a;
uint64_t b;
a |= b;
```
Instruction-level parallelism

Pipelining:
Can start to process B before finished with A (if independent)
Instruction-level parallelism

Superscalar execution: multiple parallel units, process A and B simultaneously (if independent)
Instruction-level parallelism

Out-of-order execution: run whatever you can

B depends on A,
A and C can be pipelined,
A and D use different units

code: A B C D
Instruction-level parallelism

How to exploit: make sure as much is independent as possible

Then (and only then) the CPU will figure out an efficient way to run your code
### Bad

```c
a1 *= a0;
a2 *= a1;
a3 *= a2;
a4 *= a3;
a5 *= a4;
a6 *= a5;
a7 *= a6;
a8 *= a7;
```

### Good

```c
b1 *= a1;
b2 *= a2;
b3 *= a3;
b4 *= a4;
b5 *= a5;
b6 *= a6;
b7 *= a7;
b8 *= a8;
```
Bad

\[
a1 = v[a0]; \\
a2 = v[a1]; \\
a3 = v[a2]; \\
a4 = v[a3]; \\
a5 = v[a4]; \\
a6 = v[a5]; \\
a7 = v[a6]; \\
a8 = v[a7];
\]

Good

\[
b1 = v[a1]; \\
b2 = v[a2]; \\
b3 = v[a3]; \\
b4 = v[a4]; \\
b5 = v[a5]; \\
b6 = v[a6]; \\
b7 = v[a7]; \\
b8 = v[a8];
\]
Vector instructions

• 256-bit wide “AVX” registers
  • YMM0, YMM1, …, YMM15

• Can be interpreted e.g. as:
  • a vector with $8 \times 32$-bit floats
  • a vector with $4 \times 64$-bit doubles
Vector instructions

• **SIMD** = single instruction, multiple data

• Same operation for each vector element
  - $a[i] = b[i] + c[i]$ for each $i = 0, 1, \ldots, 7$
  - $a[i] = b[i] / c[i]$ for each $i = 0, 1, \ldots, 7$

• Special functional units, special instructions
Vector instructions

float a[8]; float b[8]; float c[8];
for (int i = 0; i < 8; ++i) {
    a[i] = b[i] * c[i];
}

typedef float float8_t __attribute__ (((__vector_size__ (8*sizeof(float))));
float8_t a; float8_t b; float8_t c;
a = b * c;
Vector instructions

```c
float a[8]; float b[8]; float c[8];
for (int i = 0; i < 8; ++i) {
    a[i] = b[i] * c[i];
}

typedef float float8_t __attribute__((__vector_size__ (8*sizeof(float))));
float8_t a; float8_t b; float8_t c;
a = b * c;
```

vmulps %xmm0, %xmm1, %xmm2
Vector instructions

```c
float a[8]; float b[8]; float c[8];
for (int i = 0; i < 8; ++i) {
    a[i] = b[i] * c[i];
}

typedef float float8_t __attribute__((__vector_size__ (8*sizeof(float))));
float8_t a; float8_t b; float8_t c;
a = b * c;
```

Some care needed with **memory alignment**!
More about this later…
Multiple threads

- Multiple processors
  - independent processors
  - shared main memory
Multiple threads

- Multiple processors

- Multiple cores per processor
  - multiple processors in single package
  - often some shared components, e.g. caches
Multiple threads

- Multiple processors
- Multiple cores per processor
- Multiple threads per core
  - “Hyper-threading”
  - better utilisation of CPU resources
Multiple threads

- **My phone:** 1 processor \( \times \) 4 cores \( \times \) 1 thread
- **My laptop:** 1 processor \( \times \) 2 cores \( \times \) 2 threads
- **Our classroom:** 1 processor \( \times \) 4 cores \( \times \) 2 threads
- **CSC servers:** 2 processors \( \times \) 12 cores \( \times \) 1 thread
Multiple threads

• How to exploit:
  • run multiple processes simultaneously (e.g.: same program, different parameters)
  • OpenMP: #pragma omp
  • pthread_create(), fork(), etc.
GPU

• GPGPU: general-purpose computing on graphics processing units
  • also possible: CPU and GPU in parallel
  • also possible: multiple GPUs in parallel

• How to exploit: CUDA, OpenCL
Bit-level parallelism

Instruction-level parallelism

SIMD: vector instructions

Multiple threads

GPU

GPU + CPU in parallel

long words

automatic

vector types

OpenMP

CUDA

CUDA