Performance Analysis and Optimization

Giannis Koutsou
LAP2015, Oct. 14-16, Jülich

The Cyprus Institute

Resources

- **Reference card for OpenMP**
  - http://openmp.org/mp-documents/OpenMP3.1-CCard.pdf
- **Reference for Vector-Intrinsics**
  - https://software.intel.com/sites/landingpage/IntrinsicsGuide/
- **Exercises**
  - github.com/g-koutsou/LAP2015/

```
git clone https://github.com/g-koutsou/LAP2015.git
```

Exercises

- **Generic instructions:**
  - A Makefile for each exercise subdirectory
    - Type `make` to compile
  - A job script, `run.sh`, requesting 1 node for 5-15 mins.
    - Type `qsub run.sh` to submit job
    - Type `qstat –u $USER` to see the status (queued, running, completing)
    - Upon completion a file with a .log extension contains the output of the run
  - `__TODO__` tags in code indicate where you need to add code for completing the exercises

Exercise 1

- **Exercise 1**
  - Subdirs: `Ex1a, Ex1b, Ex1c`
- **Ex1a**
  - Complex, single precision, axpy kernel:

```
for(i=0; i<L; i++)
    y[i] = a*x[i] + y[i]
```

  - Modify program to report
    - Time per kernel call (μsec)
    - Floating point rate (Gflop/sec)
    - I/O rate (Gbytes/sec)

Exercise 1a

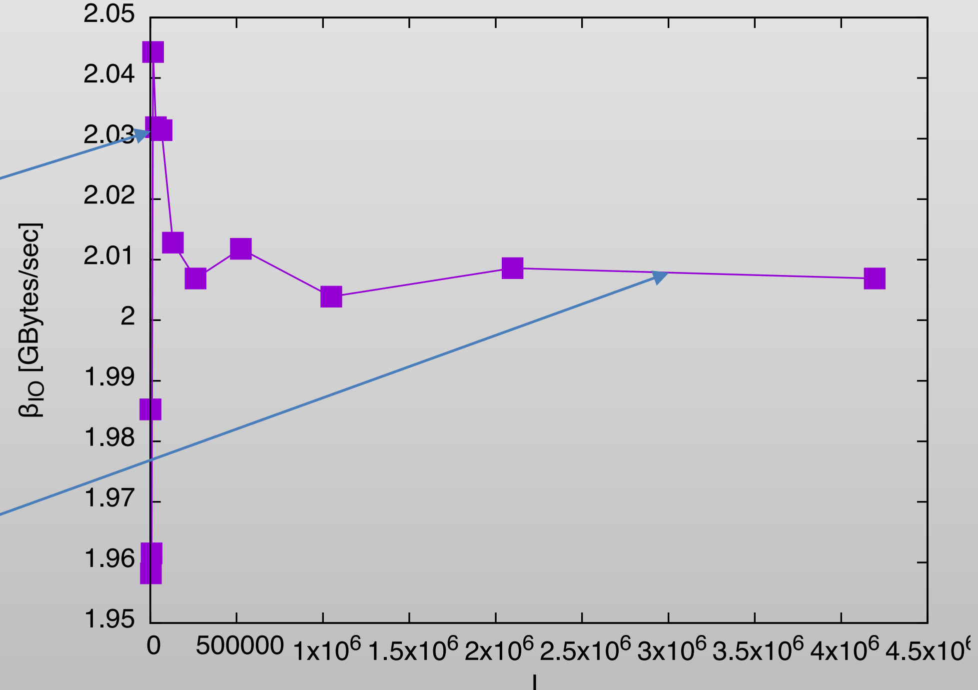
- **For example:**

```
L = 1048576, 1.26e+04±3.53e+00 usec/call, perf. = 0.668 GFlop/sec, bw = 2.004 GBytes/sec
```

- **Plot BW vs L:**

L<<, kernel arguments stay in cache

Sustained BW from RAM



Exercise 1b

- **Implement Kernel using vector-intrinsics**

```
for(i=0; i<L; i+=2) {
  y[i  ].re = a.re*x[i  ].re - a.im*x[i  ].im + y[i  ].re
  y[i  ].im = a.re*x[i  ].im + a.im*x[i  ].re + y[i  ].im
  y[i+1].re = a.re*x[i+1].re - a.im*x[i+1].im + y[i+1].re
  y[i+1].im = a.re*x[i+1].im + a.im*x[i+1].re + y[i+1].im
}
```

Exercise 1b

- **Implement Kernel using vector-intrinsics**

```
for(i=0; i<L; i+=2) {
  y[i  ].re = a.re*x[i  ].re - a.im*x[i  ].im + y[i  ].re
  y[i  ].im = a.re*x[i  ].im + a.im*x[i  ].re + y[i  ].im
  y[i+1].re = a.re*x[i+1].re - a.im*x[i+1].im + y[i+1].re
  y[i+1].im = a.re*x[i+1].im + a.im*x[i+1].re + y[i+1].im
}
```

Exercise 1b

- **Implement Kernel using vector-intrinsics**

```
for(i=0; i<L; i+=2) {
  y[i  ].re = a.re*x[i  ].re − a.im*x[i  ].im + y[i  ].re
  y[i  ].im = a.re*x[i  ].im + a.im*x[i  ].re + y[i  ].im
  y[i+1].re = a.re*x[i+1].re − a.im*x[i+1].im + y[i+1].re
  y[i+1].im = a.re*x[i+1].im + a.im*x[i+1].re + y[i+1].im
}
```

Exercise 1b

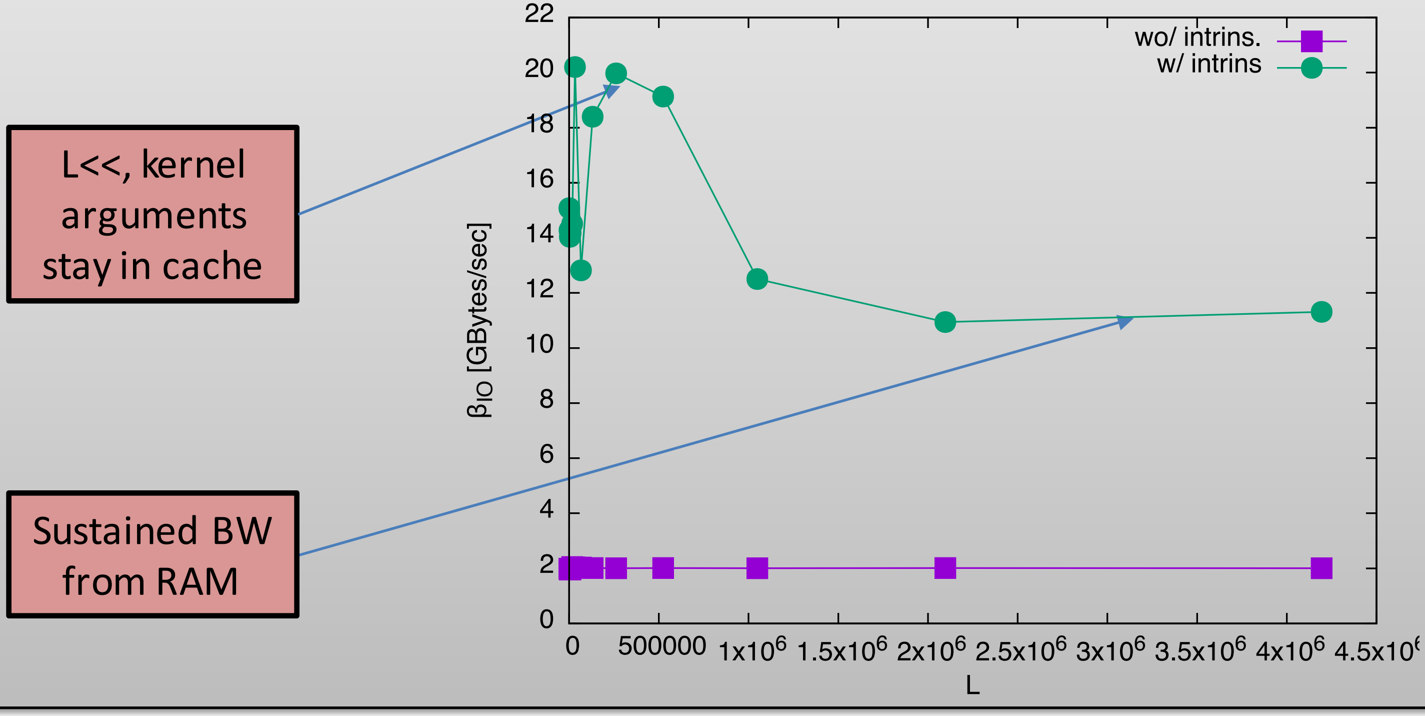
- **Implement Kernel using vector-intrinsics**

```
for(i=0; i<L; i+=2) {
  y[i  ].re = a.re*x[i  ].re - a.im*x[i  ].im + y[i  ].re
  y[i  ].im = a.re*x[i  ].im + a.im*x[i  ].re + y[i  ].im
  y[i+1].re = a.re*x[i+1].re - a.im*x[i+1].im + y[i+1].re
  y[i+1].im = a.re*x[i+1].im + a.im*x[i+1].re + y[i+1].im
}
```

- **For every (double-) iteration**
  - One `shuffle`
  - Two `add`
  - Two `mul`
  - Two `loads`, one `store`

Exercise 1b

- **Plot BW vs L:**

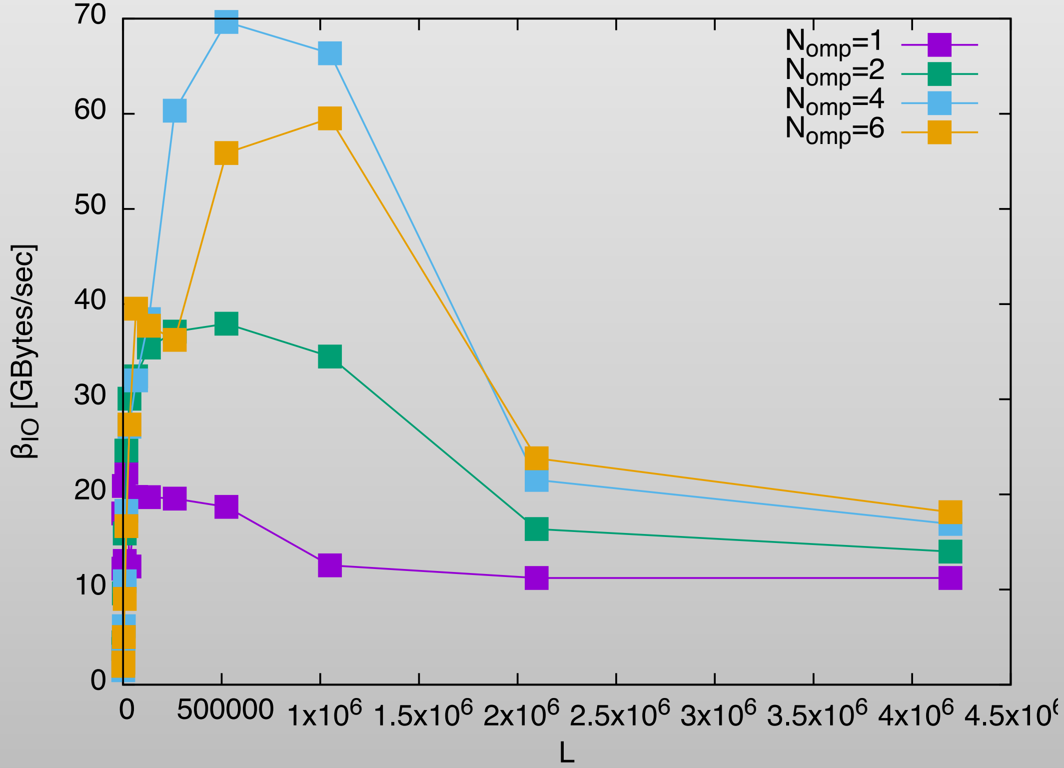


Exercise 1c

- **Add OMP pragmas to parallelize the i-loop**
- **Use:**
  - ```
    #pragma omp parallel
    {
      …
    }
    ```
  - `#pragma omp for`
- **Which variables need to be made local (private)?**
    ```

Exercise 1c

- **Plot BW vs L, for 1, 2, 4, and 6 OMP threads:**



Exercise 2

- **Ex2**
  - Array of double NxN matrices times constant NxN matrix, `mxam`:

```
for(i=0; i<L; i++)
  for(a=0; a<N; a++)
    for(b=0; b<N; b++)
      for(c=0; c<N; c++)
         y[i][a][b] += A[a][c]*x[i][c][b]
```

  - Purpose is **not** to optimise this kernel
  - This kernel will allow us to explore the transition between computational-bound and bandwidth-bound kernels (how?)

Exercise 2

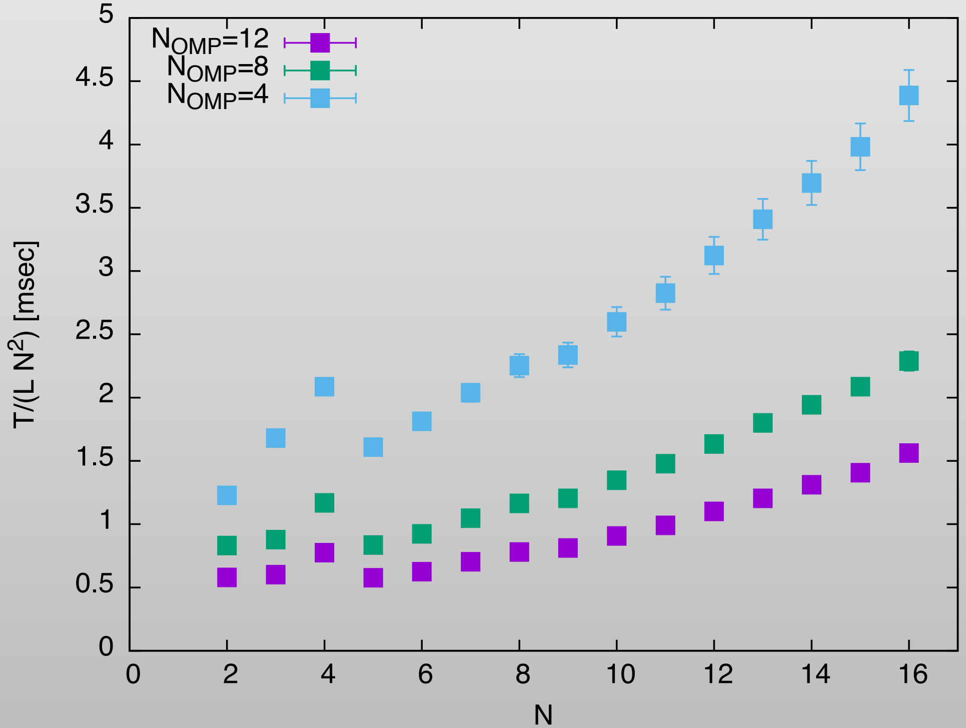
- **Ex2**
  - Array of double NxN matrices times constant NxN matrix, mxam:

```
for(i=0; i<L; i++)
  for(a=0; a<N; a++)
    for(b=0; b<N; b++)
      for(c=0; c<N; c++)
         y[i][a][b] += A[a][c]*x[i][c][b]
```

  - As in Ex1, modify program to report
    - Time per kernel call (μsec)
    - Floating point rate (Gflop/sec)
    - I/O rate (Gbytes/sec)
  - Run and plot: `T/(N**2*L)` vs `N`

Exercise 2

- **Ex2**
  - Run and plot $T/(N^2L)$ vs N



$$T_{\mathrm{FP}} = N^2 L(2N-1)\frac{1}{P\beta_{\mathrm{FP}}}$$

$$T_{\mathrm{IO}} = N^2 L\frac{1}{\beta_{\mathrm{IO}}(P)}$$

$$\frac{T}{N^2 L} = \max[\frac{2N-1}{P\beta_{\mathrm{FP}}}, \frac{1}{\beta_{\mathrm{IO}}(P)}]$$

Results may vary…

Exercise 3

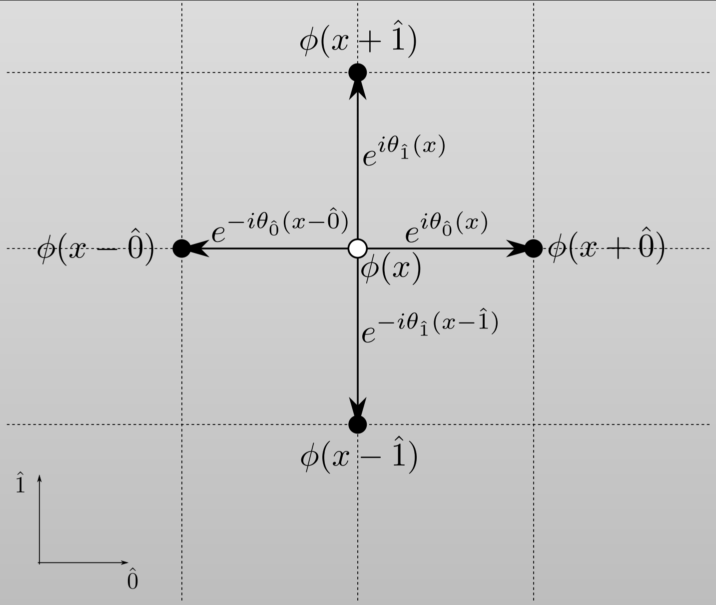
- **Ex3**
  - Laplacian stencil, with U(1) coupling

$$\boxed{\phi'(x) \leftarrow 4\phi(x) - \sum_{\mu}[\phi(x+\hat{\mu})u_{\hat{\mu}}(x) + \phi(x-\hat{\mu})u^{*}_{\hat{\mu}}(x-\hat{\mu})]}$$

$$\phi(x) \in \mathbb{C}$$

$$u_{\hat{\mu}}(x) \in U(1)$$

$$\text{2D case} \Rightarrow \mu = 0,\ 1$$



Exercise 3

- **Ex3a**

```
for(int y=0; y<L; y++)
  for(int x=0; x<L; x++) {
    int v00 = y*L + x;
    int v0p = y*L + (x+1)%L;
    int vp0 = ((y+1)%L)*L + x;
    int v0m = y*L + (L+x-1)%L;
    int vm0 = ((L+y-1)%L)*L + x;

    _Complex float p;
    p  = phi_in[v0p]*u[UIDX(v00, 0)];
    p += phi_in[vp0]*u[UIDX(v00, 1)];
    p += phi_in[v0m]*conj(u[UIDX(v0m, 0)]);
    p += phi_in[vm0]*conj(u[UIDX(vm0, 1)]);
    phi_out[v00] = 4*phi_in[v00] - p;
  }
```

Exercise 3

- **Ex3b**
  - Re-order data into a vectorisable form
  - Need 4 consecutive elements on which the same operation will be carried out