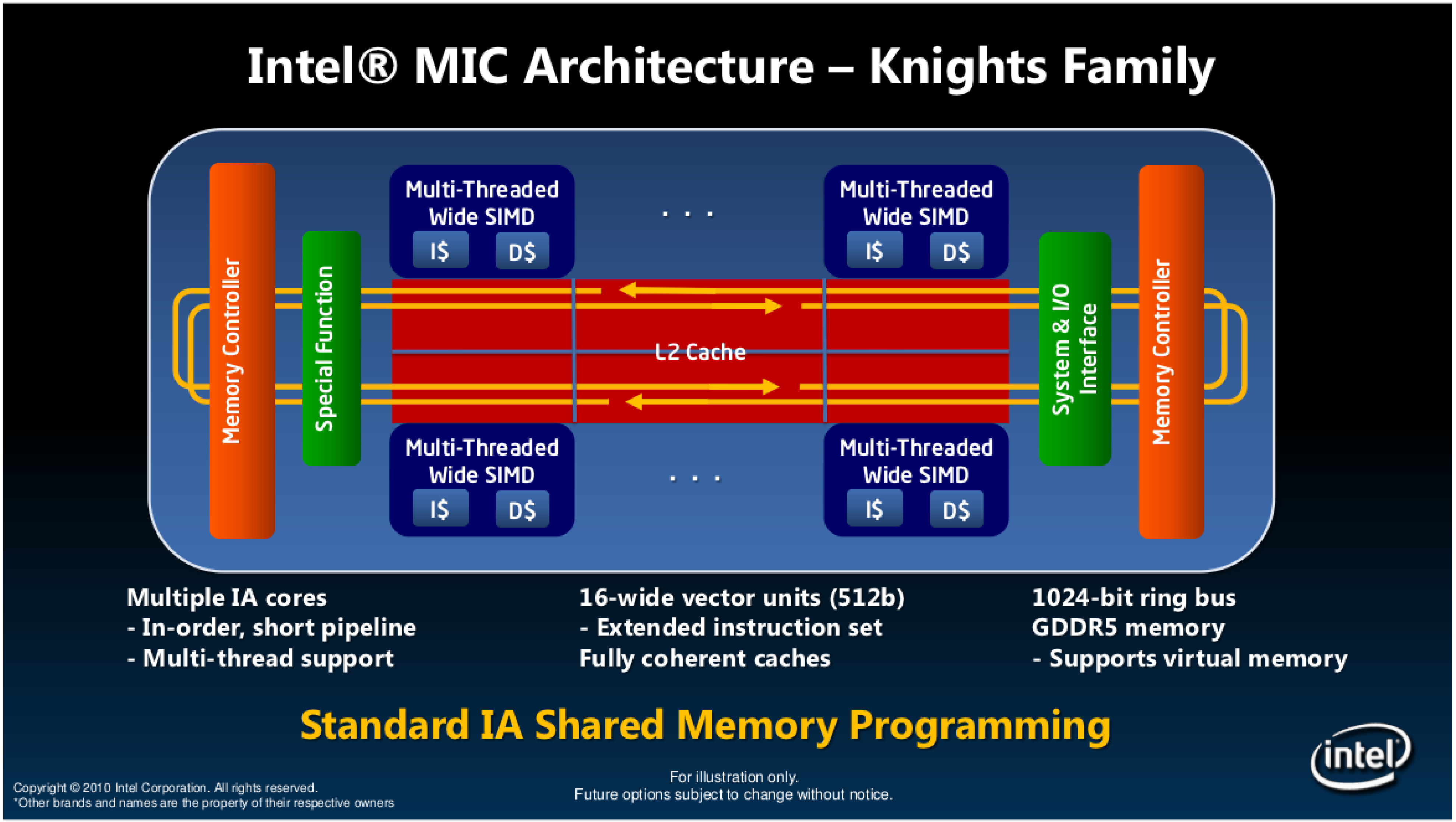
Prime esperienze con il processore Intel MIC

Francesco Giacomini – CNAF

Workshop congiunto INFN CCR – GARR
Napoli, 14-17 maggio 2012

Outline

- Introduction to the **M**any-**I**ntegrated **C**ore Architecture
- Goal of the exercise
- Findings
  - MIC-related, performance-unrelated
- Other findings
  - MIC-unrelated, performance-related



Intel® MIC Architecture – Knights Family

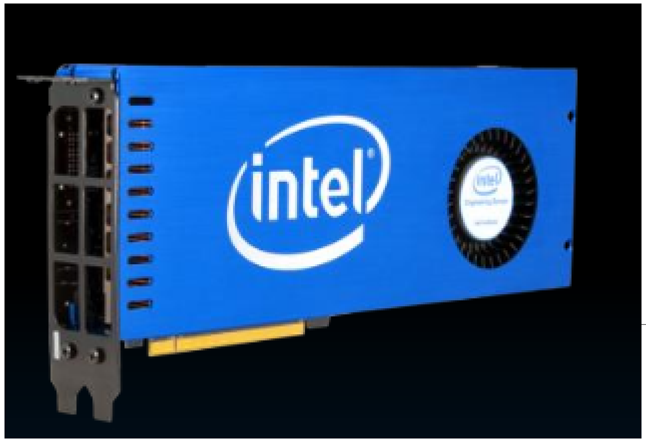
Multiple IA cores
- In-order, short pipeline
- Multi-thread support

16-wide vector units (512b)
- Extended instruction set
Fully coherent caches

1024-bit ring bus
GDDR5 memory
- Supports virtual memory

Standard IA Shared Memory Programming

For illustration only.
Future options subject to change without notice.



Ideal for parallel/vectorized code, but works for sequential code

Goal of the exercise

- Play with the MIC
  - In terms of software development
  - Contribution to the INFN COKA project (**CO**mputation on **K**nigths **A**rchitecture)
- Non-goal: measure performance
- Target is (part of) EvtGen
  - An event generator designed for the simulation of the physics of B decays
  - http://www.slac.stanford.edu/~lange/EvtGen/
  - Limited to the function EvtBtoXsgammaKagan::computeHadronicMass()
- First step to understand if and under which conditions MIC is suitable for HEP software
- Longer-term goal is to smoothly integrate the possibility to offload computation to an accelerator (such as a MIC or a GPU) directly in the software framework of an experiment

Two approaches

1. Native compilation
   - The whole application is compiled only for the MIC Instruction Set
   - The application needs to be moved explicitly onto the MIC processor before execution
2. Heterogeneous compilation
   - Some parts of the code are marked appropriately and compiled both for the host IS and the MIC IS
   - The application starts on the host and the runtime moves code and data to the MIC processor, if available, when needed
   - Used in this exercise

Target code (simplified)

- A loop with independent iterations

```
Parameters p = …;
std::vector<double> out(size);
for (int i = 0; i < size; ++i) {
  …
  TwoCoeffFcn<…> s77f(s77FermiFunction, p, …);
  SimpsonIntegrator<…> s77i(s77f, p, …);
  out[i] = s77i.evaluate(p, …);
  …
}
```

Target code (simplified)

- A loop with independent iterations
  - They can run in parallel, e.g. with openmp

```
Parameters p = …;
std::vector<double> out(size);
#pragma omp parallel for
for (int i = 0; i < size; ++i) {
  …
  TwoCoeffFcn<…> s77f(s77FermiFunction, p, …);
  SimpsonIntegrator<…> s77i(s77f, p, …);
  out[i] = s77i.evaluate(p, …);
  …
}
```

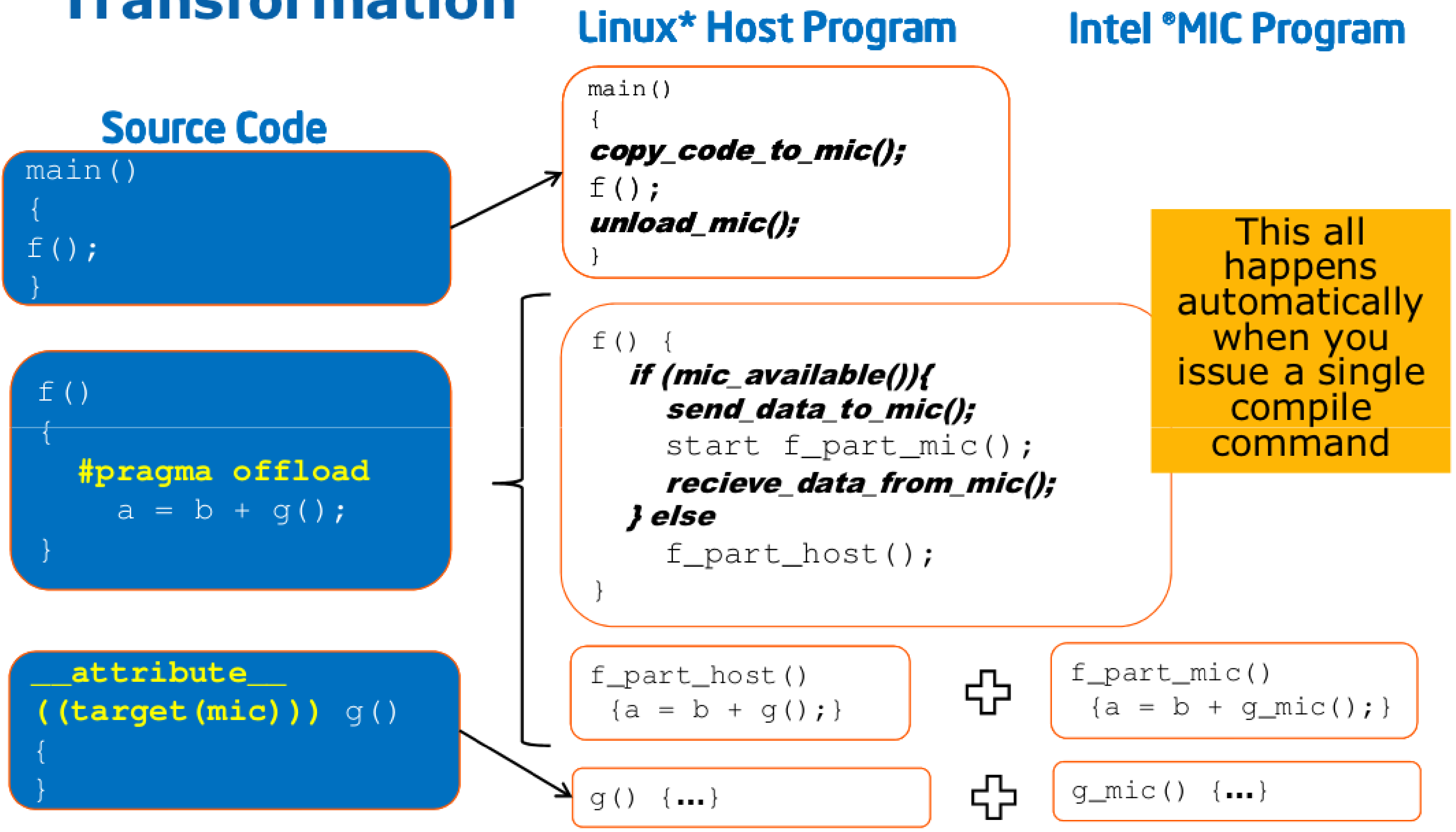
Target code (simplified)

- A loop with independent iterations
  - They can run in parallel, e.g. with openmp

```
Parameters p = …;
std::vector<double> out(size);
#pragma omp parallel for
for (int i = 0; i < size; ++i) {
  …
  TwoCoeffFcn<…> s77f(s77FermiFunction, p, …);
  SimpsonIntegrator<…> s77i(s77f, p, …);
  out[i] = s77i.evaluate(p, …);
  …
}
```

} Offload the execution onto the MIC

Heterogeneous Compiler – Conceptual Transformation



Source Code

```
main()
{
f();
}
```

```
f()
{
   #pragma offload
      a = b + g();
}
```

```
__attribute__
((target(mic))) g()
{
}
```

Linux* Host Program

```
main()
{
copy_code_to_mic();
f();
unload_mic();
}
```

```
f() {
    if (mic_available()){
       send_data_to_mic();
       start f_part_mic();
       recieve_data_from_mic();
    } else
       f_part_host();
}
```

```
f_part_host()
  {a = b + g();}
```

```
g() {…}
```

Intel ®MIC Program

```
f_part_mic()
  {a = b + g_mic();}
```

```
g_mic() {…}
```

This all happens automatically when you issue a single compile command

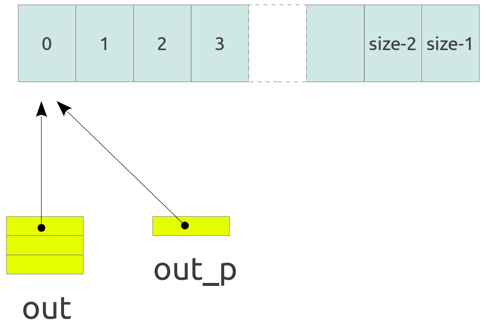
Code changes (main block)

```
Parameters p = …;
std::vector<double> out(size);
#pragma omp parallel for
for (int i = 0; i < size; ++i) {
  …
  TwoCoeffFcn<…> s77f(s77FermiFunction, p, …);
  SimpsonIntegrator<…> s77i(s77f, p, …);
  out[i] = s77i.evaluate(p, …);
  …
}
```

$\Big\}$ Offload the execution onto the MIC

Code changes

```
Parameters p = …;
std::vector<double> out(size);
double* out_p = &out[0];
#pragma offload target(mic) \
  in(p), out(out_p: length(size))
#pragma omp parallel for
for (int i = 0; i < size; ++i) {
  …
  TwoCoeffFcn<…> s77f(s77FermiFunction, p, …);
  SimpsonIntegrator<…> s77i(s77f, p, …);
  out_p[i] = s77i.evaluate(p, …);
  …
}
```



Code changes (simple functions)

```
double FermiFunc(…) {
  return …;
}

double s77(…) {
  return …;
}

double s77FermiFunction(…) {
  return FermiFunc(…) * s77(…);
}
```

Code changes (simple functions)

```
#ifdef __INTEL_OFFLOAD
#define EVT_TARGET_MIC __attribute__((target(mic)))
#else
#define EVT_TARGET_MIC
#endif

EVT_TARGET_MIC double FermiFunc(…) {
  return …;
}

EVT_TARGET_MIC double s77(…) {
  return …;
}

EVT_TARGET_MIC double s77FermiFunction(…) {
  return FermiFunc(…) * s77(…);
}
```

Code changes (entire classes)

```
template<class F>
class TwoCoeffFcn
{
  …
};
```

Code changes (entire classes)

```
#pragma offload_attribute(push, target(mic))

template<class F>
class TwoCoeffFcn
{
  …
};

#pragma offload_attribute(pop)
```

Findings

- It works!
- Not very intrusive at source code level
  - But the Intel compiler suite is needed to produce an executable that can be offloaded
- Next steps:
  - Identify a piece of code better suited to be parallelized
  - Measure performance

Other findings

- The code has been heavily restructured to make it more parallel-friendly
  - Value-based, possibly const
    - Use only stack-based objects/variables, i.e. no pointers
  - A lot less sharing between loop iterations
- Significant performance improvements as a side-effect

Changes affecting performance

- Reuse results of log, exp and pow computations

```
double l = … * log(xt) * log(xt);
```

Changes affecting performance

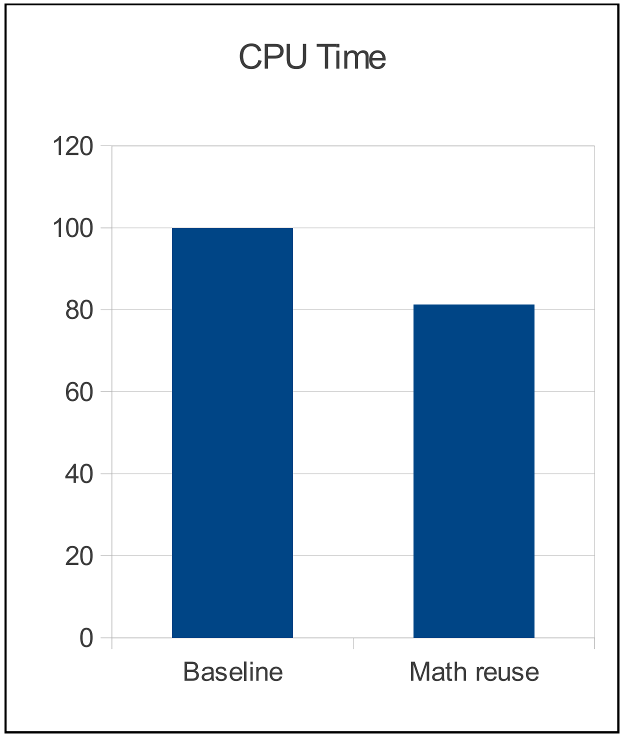
- Reuse results of log, exp and pow computations

```
double const log_xt = log(xt);
double l = … * log_xt * log_xt;
```

Changes affecting performance

- Reuse results of log, exp and pow computations

```
double const log_xt = log(xt);
double l = … * log_xt * log_xt;
```



Changes affecting performance

- Replace runtime-polymorphism (i.e. inheritance) with static-polymorphism (i.e. templates)

```
class SimpsonIntegrator: public EvtItgAbsIntegrator
{
public:
  SimpsonIntegrator(EvtItgAbsFunction const& integrand, …);
  virtual ~SimpsonIntegrator();
  …
};
```

Changes affecting performance

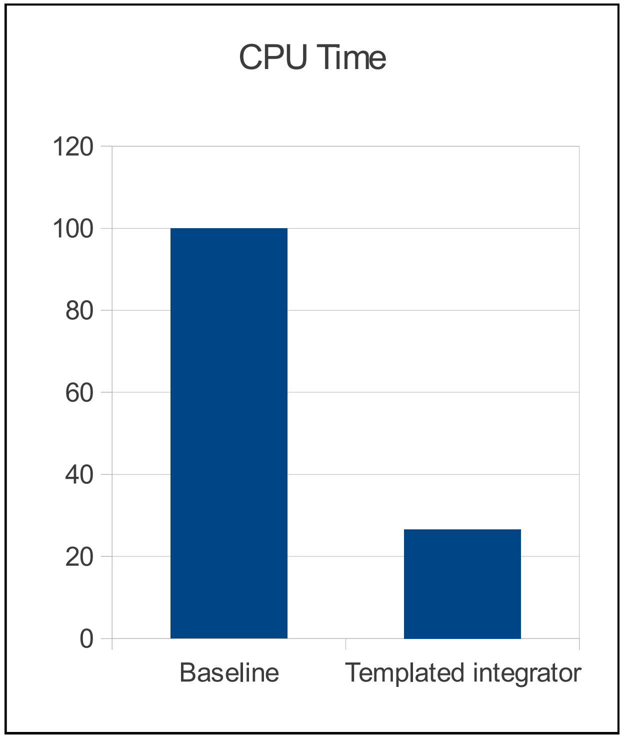
- Replace runtime-polymorphism (i.e. inheritance) with static-polymorphism (i.e. templates)

```
template<typename F>
class SimpsonIntegrator
{
public:
  SimpsonIntegrator(F const& integrand, …);
  …
};
```

Changes affecting performance

- Replace runtime-polymorphism (i.e. inheritance) with static-polymorphism (i.e. templates)

```
template<typename F>
class SimpsonIntegrator
{
public:
  SimpsonIntegrator(F const& integrand, …);
  …
};
```



Changes affecting performance

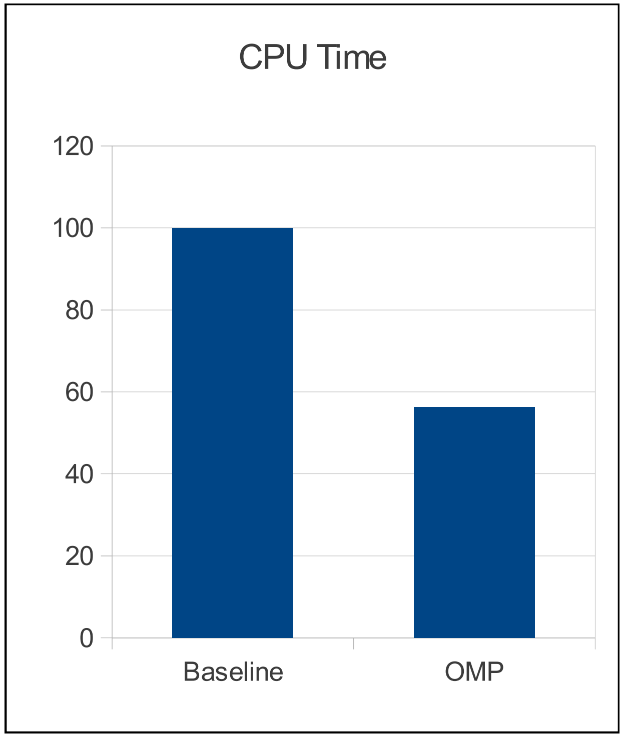
- Parallelism (two threads)

```
for (int i = 0; i < size; ++i) {
  …
}
```

Changes affecting performance

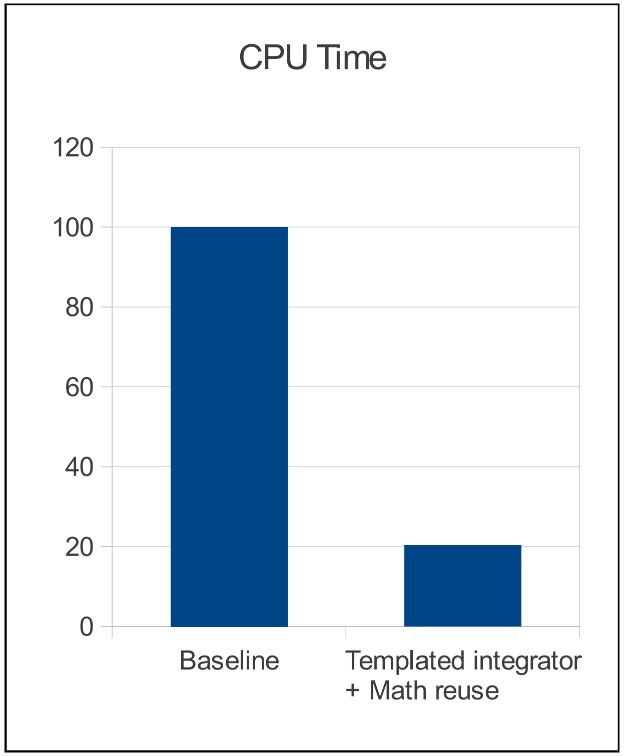
- Parallelism (two threads)

```
#pragma omp parallel for
for (int i = 0; i < size; ++i) {
   …
}
```



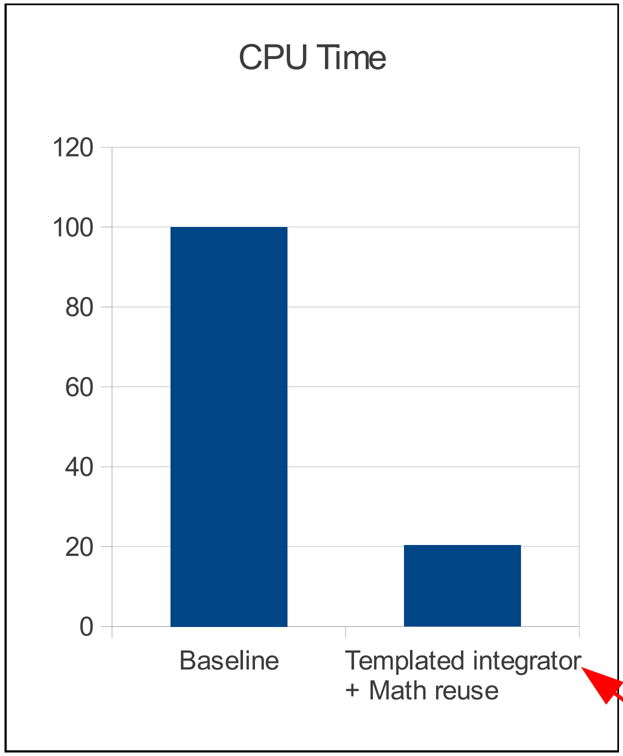
Changes affecting performance

- Combining effects



Changes affecting performance

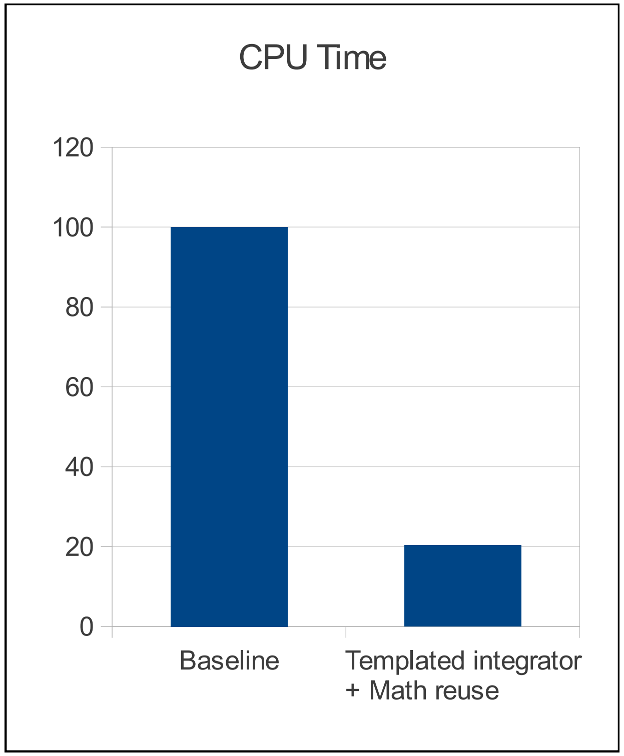
- Combining effects

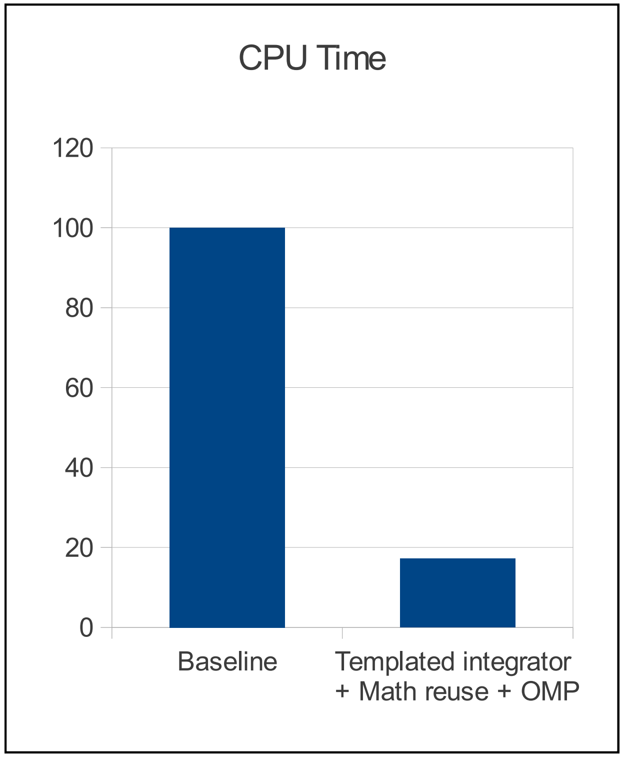


A factor 5 faster!

Changes affecting performance

- Combining effects





Conclusions

- The work on porting software to a parallel environment and to MIC in particular looks promising so far
- There is evidence that even improving existing code would bring large benefits