Experiences with OpenMP, PGI, HMPP and OpenACC directives on ISO/TTI kernels

Sayan Ghosh [1] Terrence Liao [2] Henri Calandra [2]
Barbara M. Chapman [1]

[1]University of Houston

[2]Total

16 Nov 2012

Multi-Core Computing Systems (MuCoCoS) 2012 Workshop

Sections

Accelerator directives, testbed and test kernels

Optimizations using accelerator directives and compiler options

Analysis of various CPU/GPU versions of ISO/TTI kernels

Re-evaluation of the directives

Why Accelerator Directives?

- GPUs are slowly becoming ubiquitous in HPC

Why Accelerator Directives?

- GPUs are slowly becoming ubiquitous in HPC
- GPU programming requires steep learning curve

Why Accelerator Directives?

- GPUs are slowly becoming ubiquitous in HPC
- GPU programming requires steep learning curve
- Re-writing legacy applications in CUDA/OpenCL is challenging

Why Accelerator Directives?

- GPUs are slowly becoming ubiquitous in HPC
- GPU programming requires steep learning curve
- Re-writing legacy applications in CUDA/OpenCL is challenging
- Expected - substantial speedup with minimal effort

Why Accelerator Directives?

- GPUs are slowly becoming ubiquitous in HPC
- GPU programming requires steep learning curve
- Re-writing legacy applications in CUDA/OpenCL is challenging
- Expected - substantial speedup with minimal effort
- Different accelerators have different capabilities - end user should be exposed to a uniform interface

Why Accelerator Directives?

- GPUs are slowly becoming ubiquitous in HPC
- GPU programming requires steep learning curve
- Re-writing legacy applications in CUDA/OpenCL is challenging
- Expected - substantial speedup with minimal effort
- Different accelerators have different capabilities - end user should be exposed to a uniform interface
- Improves code maintainability/portability

Accelerator directives model

- Follows OpenMP, which pioneered the usage of simple directives (*pragmas*)

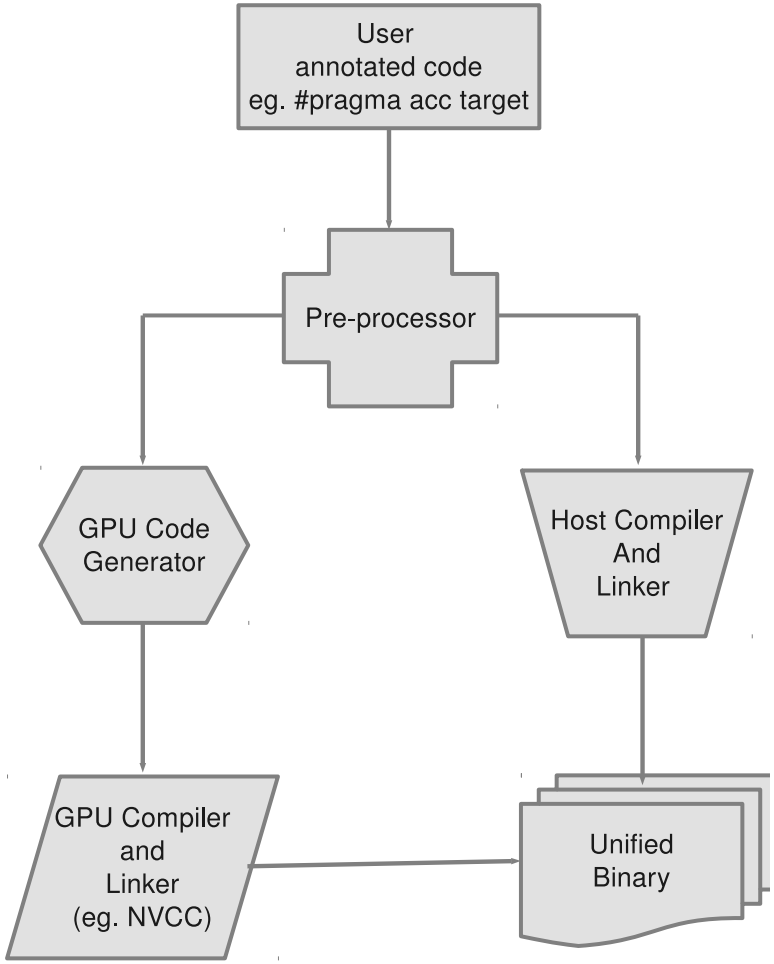
Accelerator directives model

- Follows OpenMP, which pioneered the usage of simple directives (*pragmas*)
- Abstract hardware level details from the user

Accelerator directives model

- Follows OpenMP, which pioneered the usage of simple directives (*pragmas*)
- Abstract hardware level details from the user
- Compiler translates pragma embedded user code (and **optimizes**), according to the specified target hardware

Accelerator directives model



Compilation and code generation process of a high level GPU directive based model

Accelerator directives - HMPP, PGI and OpenACC

- **Data Directives** - Copying data to/from GPU

Accelerator directives - HMPP, PGI and OpenACC

- **Data Directives** - Copying data to/from GPU
- **Kernel/Compute Directives** - Specify the portion of code (the kernel region) to be executed on the GPU

Accelerator directives - HMPP, PGI and OpenACC

- **Data Directives** - Copying data to/from GPU
- **Kernel/Compute Directives** - Specify the portion of code (the kernel region) to be executed on the GPU
- **Loop mapping/optimization directives** - Distribution/Scheduling loops to GPU grid of blocks (in X/Y direction)

Accelerator directives - HMPP, PGI and OpenACC

- **Data Directives** - Copying data to/from GPU
- **Kernel/Compute Directives** - Specify the portion of code (the kernel region) to be executed on the GPU
- **Loop mapping/optimization directives** - Distribution/Scheduling loops to GPU grid of blocks (in X/Y direction)

All of them could be merged into a single statement, of course

Kernels used in this study - ISO and TTI

- These finite difference kernels are used in RTM

Kernels used in this study - ISO and TTI

- These finite difference kernels are used in RTM
- Reverse Time Migration (RTM) is a method to model the subsurface of the earth using two-way wave equation

Kernels used in this study - ISO and TTI

- These finite difference kernels are used in RTM
- Reverse Time Migration (RTM) is a method to model the subsurface of the earth using two-way wave equation

$$\frac{1}{c^2}\frac{\partial^2 P}{\partial t^2} = \frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2} + \frac{\partial^2 P}{\partial z^2}$$

where c is a propagated wave velocity and the P is the wavefield amplitude.

- Isotropic RTM will not be able to handle anisotropic media and will produces incorrect images

Kernels used in this study - ISO and TTI

- These finite difference kernels are used in RTM
- Reverse Time Migration (RTM) is a method to model the subsurface of the earth using two-way wave equation

$$\frac{1}{c^2}\frac{\partial^2 P}{\partial t^2} = \frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2} + \frac{\partial^2 P}{\partial z^2}$$

where c is a propagated wave velocity and the P is the wavefield amplitude.

- Isotropic RTM will not be able to handle anisotropic media and will produces incorrect images
  - Hence, pseudo-acoustic wave approximation for Transversely Isotropic (TI) media

Kernels used in this study - ISO and TTI

- These finite difference kernels are used in RTM
- Reverse Time Migration (RTM) is a method to model the subsurface of the earth using two-way wave equation

$$\frac{1}{c^2}\frac{\partial^2 P}{\partial t^2} = \frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2} + \frac{\partial^2 P}{\partial z^2}$$

where c is a propagated wave velocity and the P is the wavefield amplitude.

- Isotropic RTM will not be able to handle anisotropic media and will produces incorrect images
  - Hence, pseudo-acoustic wave approximation for Transversely Isotropic (TI) media

$$\frac{\partial^2 p}{\partial t^2} = v_{px}^2 H_2 p + \alpha v_{pz}^2 H_1 q + v_{sz}^2 H_1 (p - \alpha q)$$

$$\frac{\partial^2 q}{\partial t^2} = \frac{v_{px}^2 H_2 p}{\alpha} + v_{pz}^2 H_1 q + v_{sz}^2 H_1 (\frac{1}{\alpha} p - q)$$

Intention of the study

- **Programmability** - Ease of use, features available, adherence of compilers to specifications.

Intention of the study

- **Programmability** - Ease of use, features available, adherence of compilers to specifications.
- **Adaptability** - How much of base code change is required to incorporate the directives?

Intention of the study

- **Programmability** - Ease of use, features available, adherence of compilers to specifications.
- **Adaptability** - How much of base code change is required to incorporate the directives?
- **Performance** - How is the performance w.r.t multithreaded CPU code? Does the compilers provide hints so that performance limiters could be identified?

Evaluation platform

Platform	Facets
CPU	Intel Xeon E5640 @ 2.67 GHz, 8 CPU cores, 12 MB L3 Cache, 96 GB Memory, 16X PCIe2 bus - ideal bandwidth: 8 GB/sec
GPU	Nvidia Tesla M2090, Registers per thread: 63, 512 Compute cores, 6 GB GDDR5 Memory, ECC: disabled
Compilers	Intel Compiler 12.1.5, CAPS HMPP Workbench 3.2.1, PGI Compiler 12.3 / 12.6, Nvidia CUDA 4.0 / 4.2
API	OpenMP 3.1, OpenACC 1.0, PGI Accelerator Model v1.3

ISO/TTI kernels are written in FORTRAN language

CAPS HMPP directives

- A *Codelet* signifies a region where the function to be ported to the hardware accelerator is declared
- A *Callsite* refers to the place in the program where the function is called

HMPP **Codelet** and Callsite

```
!$HMPP fdtd CODELET, TARGET=CUDA, ARGS[V;U].MIRROR, &
...
!$HMPPCG GRIDIFY (j*i,k), BLOCKSIZE 64X8, PRIVATE(a,b,c)
  do k=k0,k1
   do j=j0,j1
    do i=i0,i1
```

HMPP Codelet and **Callsite**

```
!$HMPP fdtd ALLOCATE, DATA["in";"out"], &
...
!$HMPP fdtd ADVANCEDLOAD, DATA["in";"out"]
!$HMPP fdtd CALLSITE
CALL FDTD_base (in, out, dx, dy, dz,
           c[0], c[1], c[2], c[3], c[4]);
...
!$HMPP fdtd DELEGATEDSTORE, DATA["out"]
```

PGI Accelerator directives

The data region - `!$acc data region` encapsulates the accelerator compute region (or kernel) - `$acc region`

```
!$ACC DATA REGION COPY(V,U) COPYIN(c)
!$ACC REGION
!$ACC DO PARALLEL(64) PRIVATE(a,b)
do k=k0,k1
 !$ACC DO PARALLEL(4)
 do j=j0,j1
  !$ACC DO VECTOR(128)
  do i=i0,i1
```

OpenACC directives

- For `!$acc kernels`, the compiler would break a tight loop-nest into a sequence of kernels
- The `!$acc parallel` directive is like `!$omp parallel`, and it generates one kernel
- Expresses concurrency in terms of gangs (*blocks*) of workers (*warp*) of vectors (*threads*)

```
!$ACC DATA COPY(p0,p1)  &
!$ACC COPYIN(c)
!$ACC KERNELS           &
!$ACC PRESENT(p0,p1,c)
!$ACC LOOP INDEPENDENT
do k=k0,k1
 !$ACC LOOP INDEPENDENT
 do j=j0,j1
 !$ACC LOOP INDEPENDENT
  do i=i0,i1
```

Optimizations performed

- Loop Collapsing - GRIDIFY(j*i,k) in HMPP, automatic loop collapse in HMPP OpenACC, COLLAPSE clause in PGI Acc, OpenMP COLLAPSE clause

Optimizations performed

- Loop Collapsing - GRIDIFY(j*i,k) in HMPP, automatic loop collapse in HMPP OpenACC, COLLAPSE clause in PGI Acc, OpenMP COLLAPSE clause
- Loop Unroll/Vectorization - Each GPU thread has 63 registers - substantial ILP possible; SSE in CPUs (via compiler switches)

Optimizations performed

- Loop Collapsing - GRIDIFY(j*i,k) in HMPP, automatic loop collapse in HMPP OpenACC, COLLAPSE clause in PGI Acc, OpenMP COLLAPSE clause
- Loop Unroll/Vectorization - Each GPU thread has 63 registers - substantial ILP possible; SSE in CPUs (via compiler switches)
- Loop Re-ordering - Innermost loop might drive coalesced accesses (typically the two outer most loops are distributed to GPU block indices)

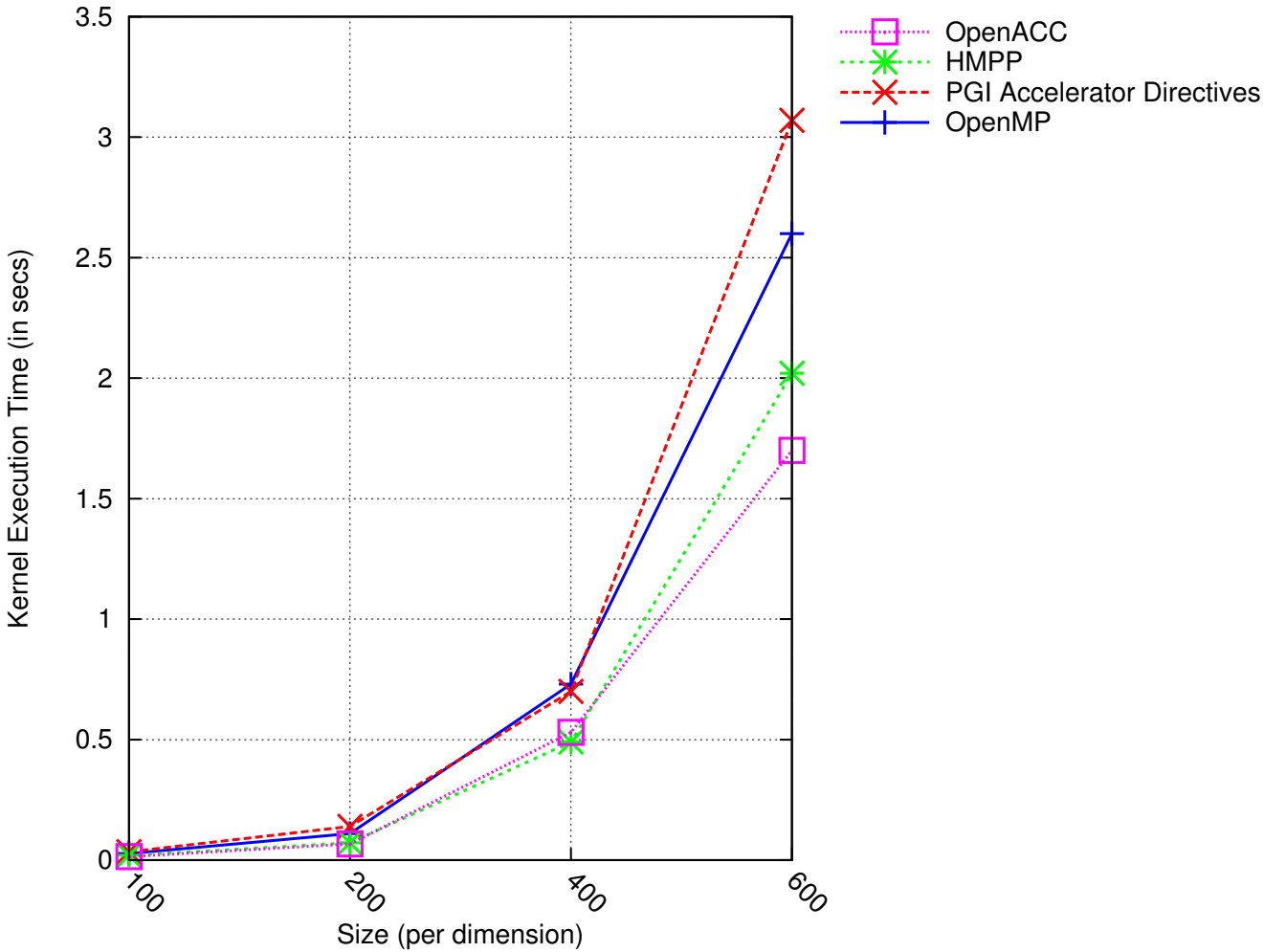
Optimizations performed

- Loop Collapsing - GRIDIFY(j*i,k) in HMPP, automatic loop collapse in HMPP OpenACC, COLLAPSE clause in PGI Acc, OpenMP COLLAPSE clause
- Loop Unroll/Vectorization - Each GPU thread has 63 registers - substantial ILP possible; SSE in CPUs (via compiler switches)
- Loop Re-ordering - Innermost loop might drive coalesced accesses (typically the two outer most loops are distributed to GPU block indices)
- Cache blocking - Beneficial for stencil operations; CACHE clause in OpenACC move data to GPU shared memory

Optimizations performed

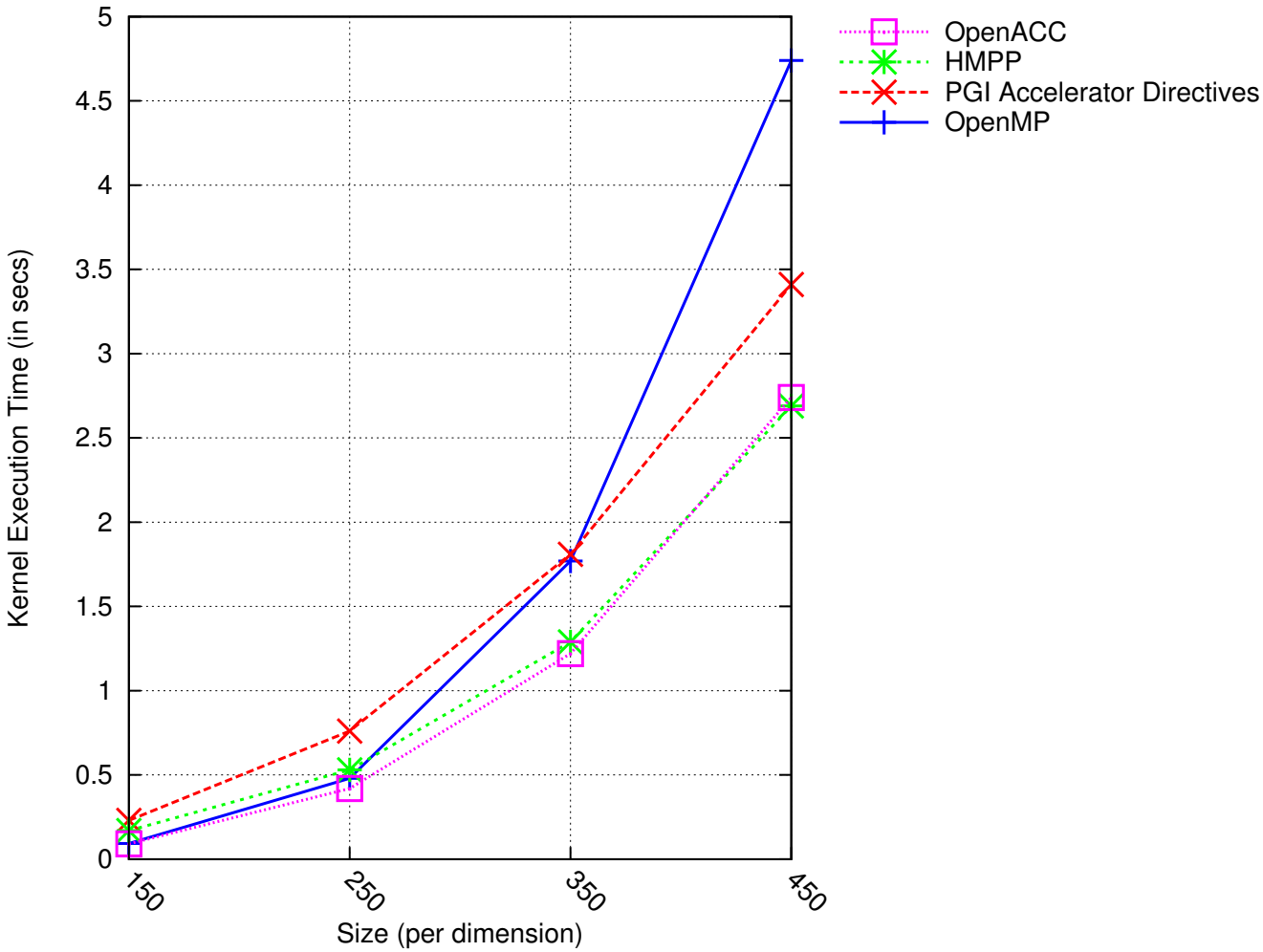
- Loop Collapsing - GRIDIFY(j*i,k) in HMPP, automatic loop collapse in HMPP OpenACC, COLLAPSE clause in PGI Acc, OpenMP COLLAPSE clause
- Loop Unroll/Vectorization - Each GPU thread has 63 registers - substantial ILP possible; SSE in CPUs (via compiler switches)
- Loop Re-ordering - Innermost loop might drive coalesced accesses (typically the two outer most loops are distributed to GPU block indices)
- Cache blocking - Beneficial for stencil operations; CACHE clause in OpenACC move data to GPU shared memory
- Asynchronous computation and Data transfers - Current/Past GPUs have two copy engines and one kernel engine

Quick peek at the final results - ISO



Different implementations of ISO using directive-based approaches

Quick peek at the final results - TTI



Different implementations of TTI using directive-based approaches

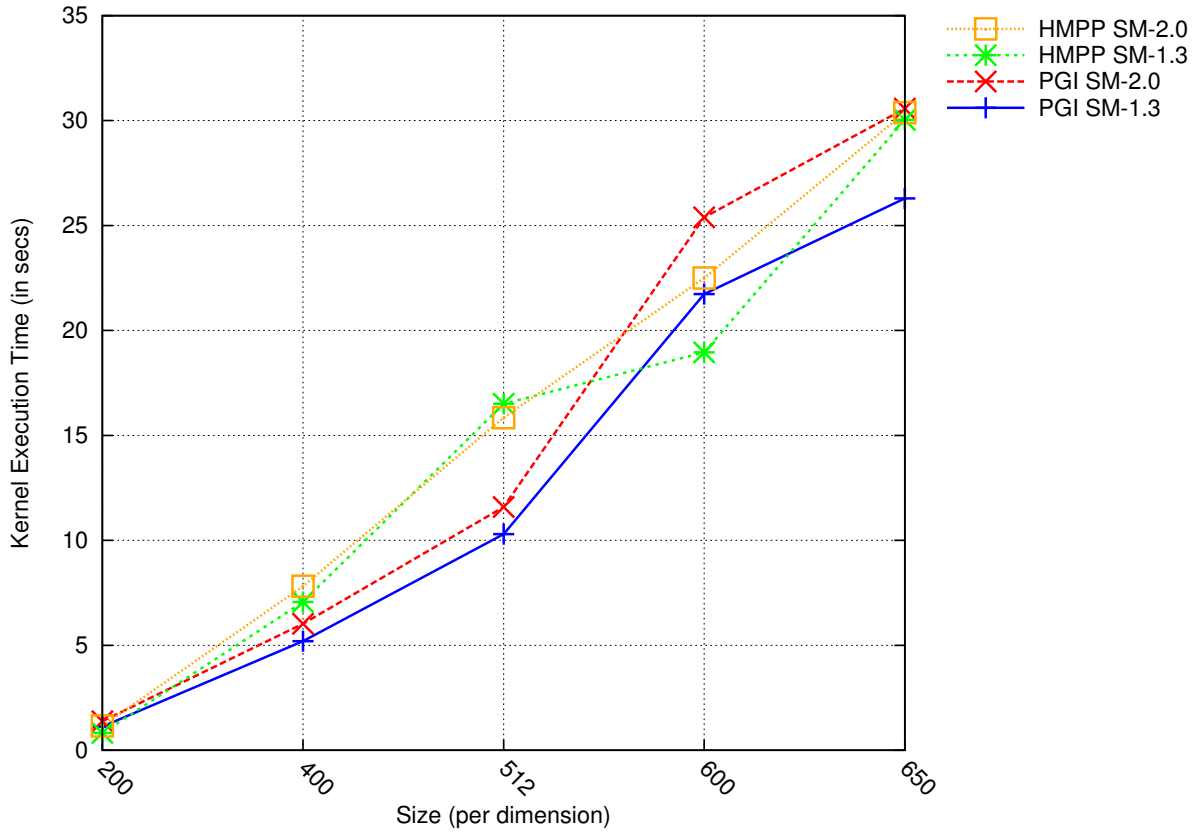
GPU acceleration factor compared to CPU implementation

Acceleration of TTI and ISO kernels compared with directive based approaches on a GPU against multi-threaded OpenMP cache-blocked implementation on an 8-core SMP

Accelerator Directives / Kernels	**ISO**	**TTI**
CAPS HMPP	1.29	1.76
PGI Accelerator	0.85	1.39
OpenACC	1.54	1.73

Differences in compute device 1.3 and 2.0

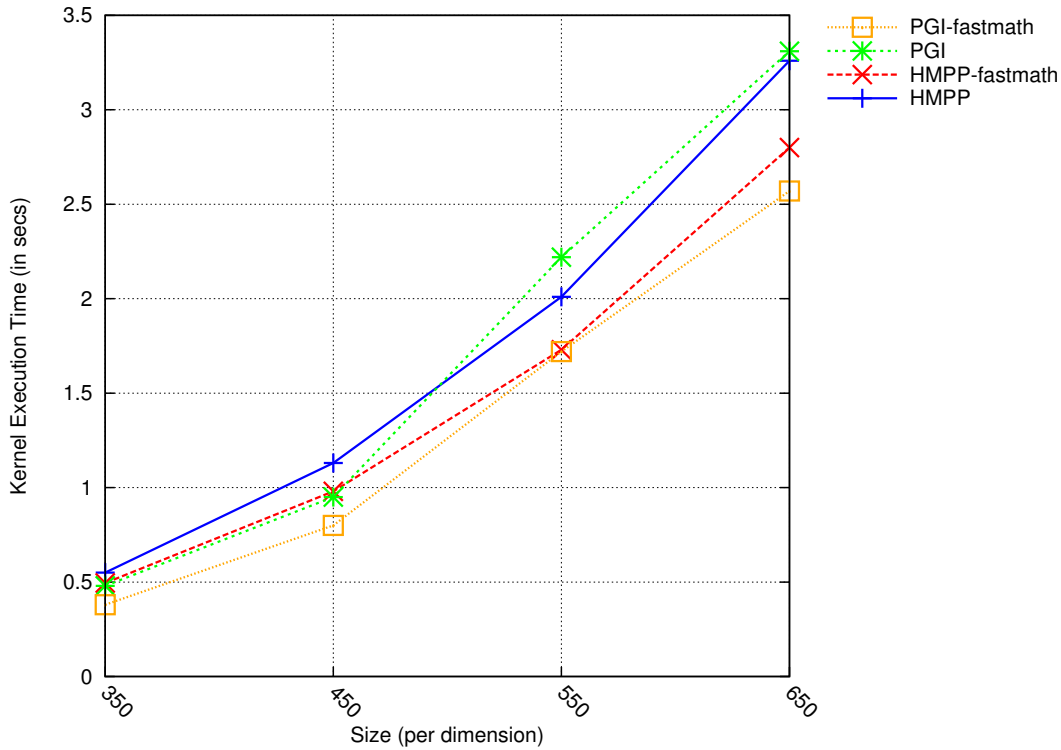
- cc13 implemented a truncated FMAD, which made it faster but inaccurate



Differences between PGI/HMPP sm13 (previous generation) vs sm20 (current generation) device options on ISO

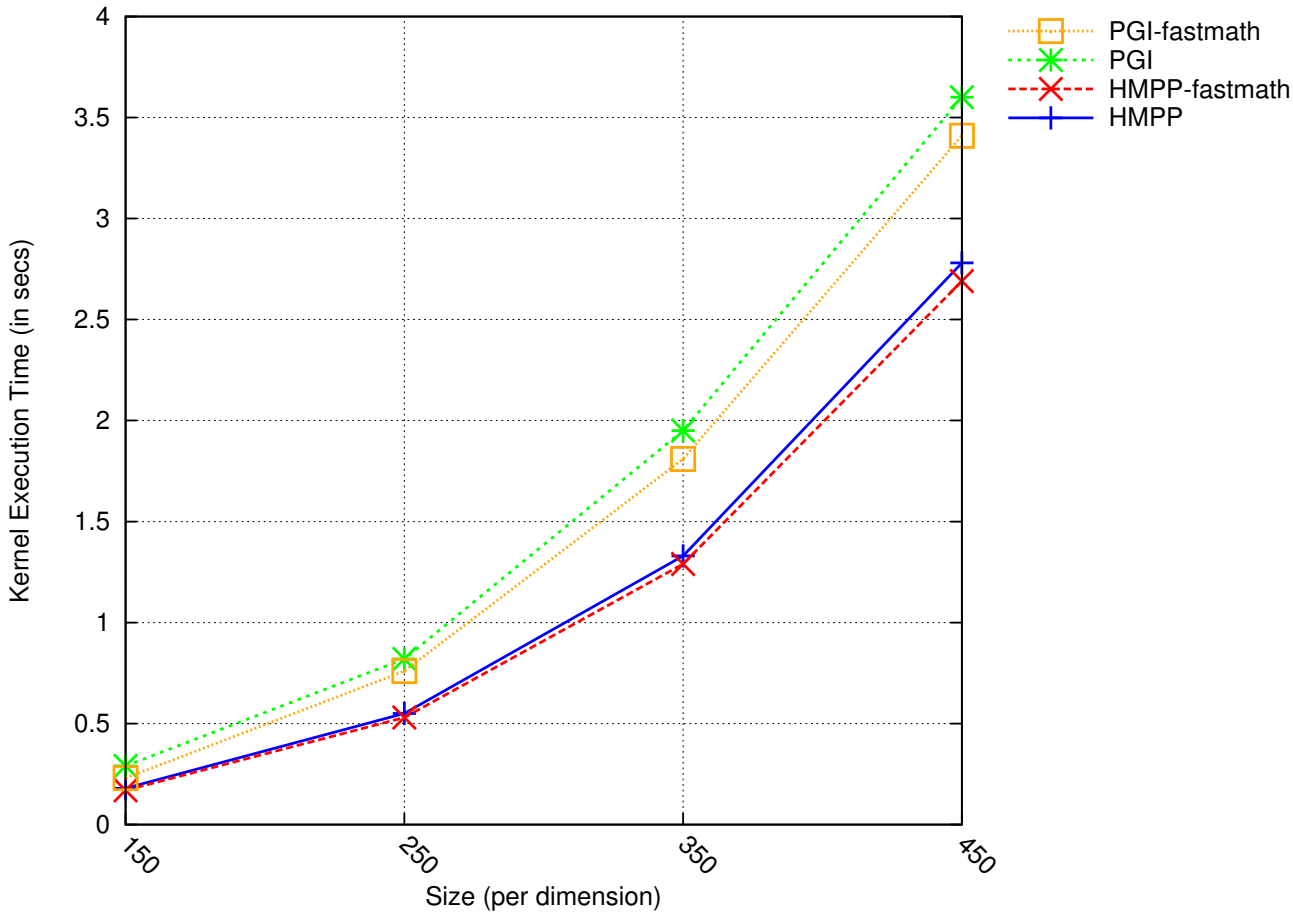
Fast-math compiler option - ISO

- HMPP: `-ftz=true -prec-div=false -prec-sqrt=false -use-fast-math`
- PGI: `-ta=nvidia,cc20,flushz,fastmath`



- This option just substitutes certain math functions with faster less-precise alternatives - independent of compiler front-ends

Fast-math compiler option - TTI



- More multiplications, more scope for fast-math optimizations

Enabling/disabling FMA

- NVCC converts all standalone multiplications to h/w specific intrinsics

Enabling/disabling FMA

- NVCC converts all standalone multiplications to h/w specific intrinsics
- Prior to cc20 - aggressive *combine and truncate operation* (Floating point multiply-add or FMAD)

Enabling/disabling FMA

- NVCC converts all standalone multiplications to h/w specific intrinsics
- Prior to cc20 - aggressive *combine and truncate operation* (Floating point multiply-add or FMAD)
  - Faster, but less accurate

Enabling/disabling FMA

- NVCC converts all standalone multiplications to h/w specific intrinsics
- Prior to cc20 - aggressive *combine and truncate operation* (Floating point multiply-add or FMAD)
  - Faster, but less accurate
- Now, FMA (Fused multiply-add) - when FMA is used: $RN(axb + c)$ - when FMA is not used: $RN(RN(axb) + c)$

Enabling/disabling FMA

- NVCC converts all standalone multiplications to h/w specific intrinsics
- Prior to cc20 - aggressive *combine and truncate operation* (Floating point multiply-add or FMAD)
  - Faster, but less accurate
- Now, FMA (Fused multiply-add) - when FMA is used: $RN(axb + c)$ - when FMA is not used: $RN(RN(axb) + c)$
- Intrinsics are not merged to FMA operations

Enabling/disabling FMA

- NVCC converts all standalone multiplications to h/w specific intrinsics
- Prior to cc20 - aggressive *combine and truncate operation* (Floating point multiply-add or FMAD)
  - Faster, but less accurate
- Now, FMA (Fused multiply-add) - when FMA is used: $RN(axb + c)$ - when FMA is not used: $RN(RN(axb) + c)$
- Intrinsics are not merged to FMA operations
- From CUDA 4.1, a switch called `fmad=true` or `fmad=false` could be passed to the compiler to control this behavior

Enabling/disabling FMA

- NVCC converts all standalone multiplications to h/w specific intrinsics
- Prior to cc20 - aggressive *combine and truncate operation* (Floating point multiply-add or FMAD)
  - Faster, but less accurate
- Now, FMA (Fused multiply-add) - when FMA is used: $RN(axb + c)$ - when FMA is not used: $RN(RN(axb) + c)$
- Intrinsics are not merged to FMA operations
- From CUDA 4.1, a switch called `fmad=true` or `fmad=false` could be passed to the compiler to control this behavior
- For TTI, turning off FMA gave better performance (5%), just the opposite for ISO

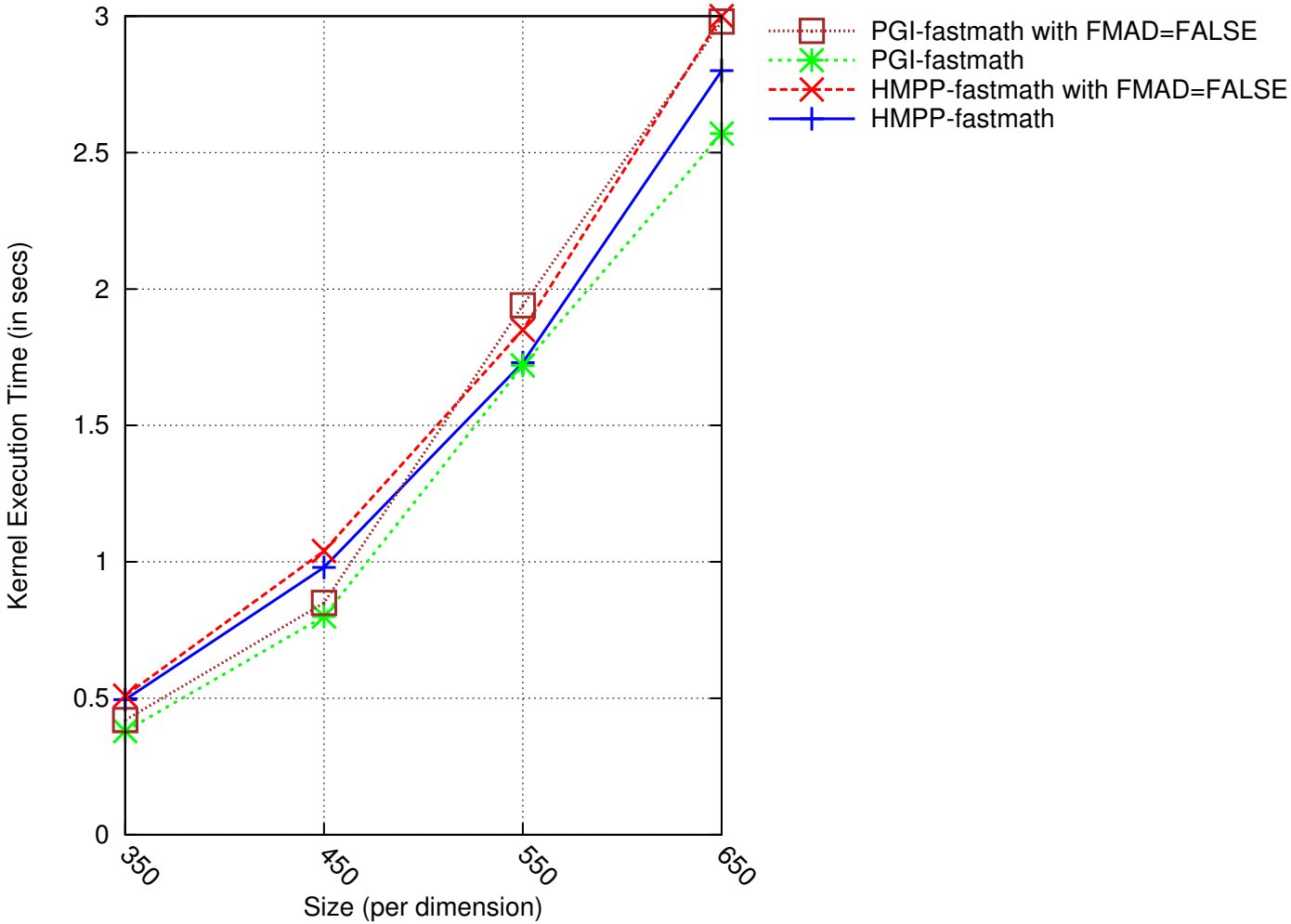
Enabling/disabling FMA

- NVCC converts all standalone multiplications to h/w specific intrinsics
- Prior to cc20 - aggressive *combine and truncate operation* (Floating point multiply-add or FMAD)
  - Faster, but less accurate
- Now, FMA (Fused multiply-add) - when FMA is used: $RN(axb + c)$ - when FMA is not used: $RN(RN(axb) + c)$
- Intrinsics are not merged to FMA operations
- From CUDA 4.1, a switch called `fmad=true` or `fmad=false` could be passed to the compiler to control this behavior
- For TTI, turning off FMA gave better performance (5%), just the opposite for ISO
  - Instructions per byte when FMA is turned OFF - 4.22 (ideal for M2090: 3.79) - instruction bound

Enabling/disabling FMA

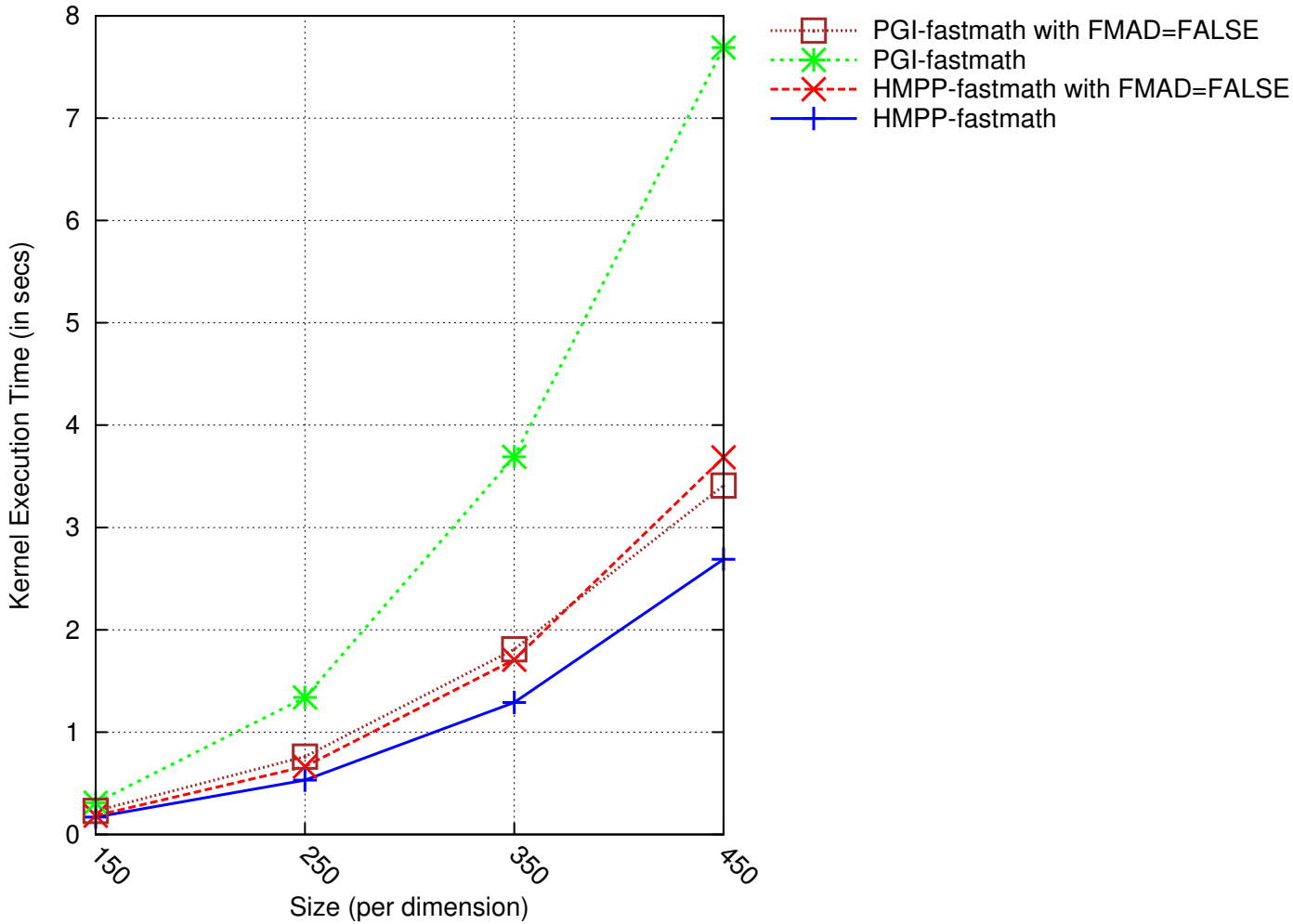
- NVCC converts all standalone multiplications to h/w specific intrinsics
- Prior to cc20 - aggressive *combine and truncate operation* (Floating point multiply-add or FMAD)
  - Faster, but less accurate
- Now, FMA (Fused multiply-add) - when FMA is used: $RN(axb + c)$ - when FMA is not used: $RN(RN(axb) + c)$
- Intrinsics are not merged to FMA operations
- From CUDA 4.1, a switch called `fmad=true` or `fmad=false` could be passed to the compiler to control this behavior
- For TTI, turning off FMA gave better performance (5%), just the opposite for ISO
  - Instructions per byte when FMA is turned OFF - 4.22 (ideal for M2090: 3.79) - instruction bound
  - Instructions per byte when FMA is turned ON - 2.79 (ideal for M2090: 3.79) - memory bound

Enabling/disabling FMA - ISO and TTI



HMPP/PGI with FMA turned OFF Vs FMA turned ON for ISO (memory bound)

Enabling/disabling FMA - ISO and TTI

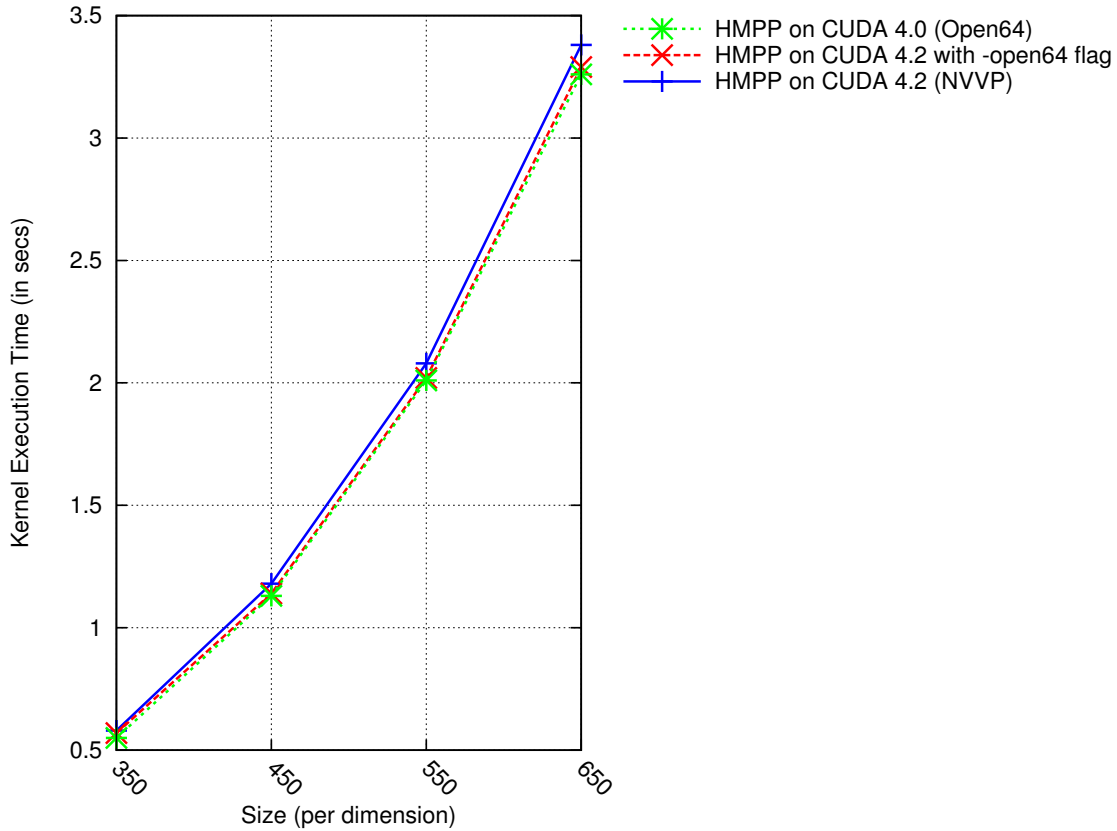


- For HMPP - performance drops when FMA=FALSE; For PGI - performance increased when FMA=FALSE

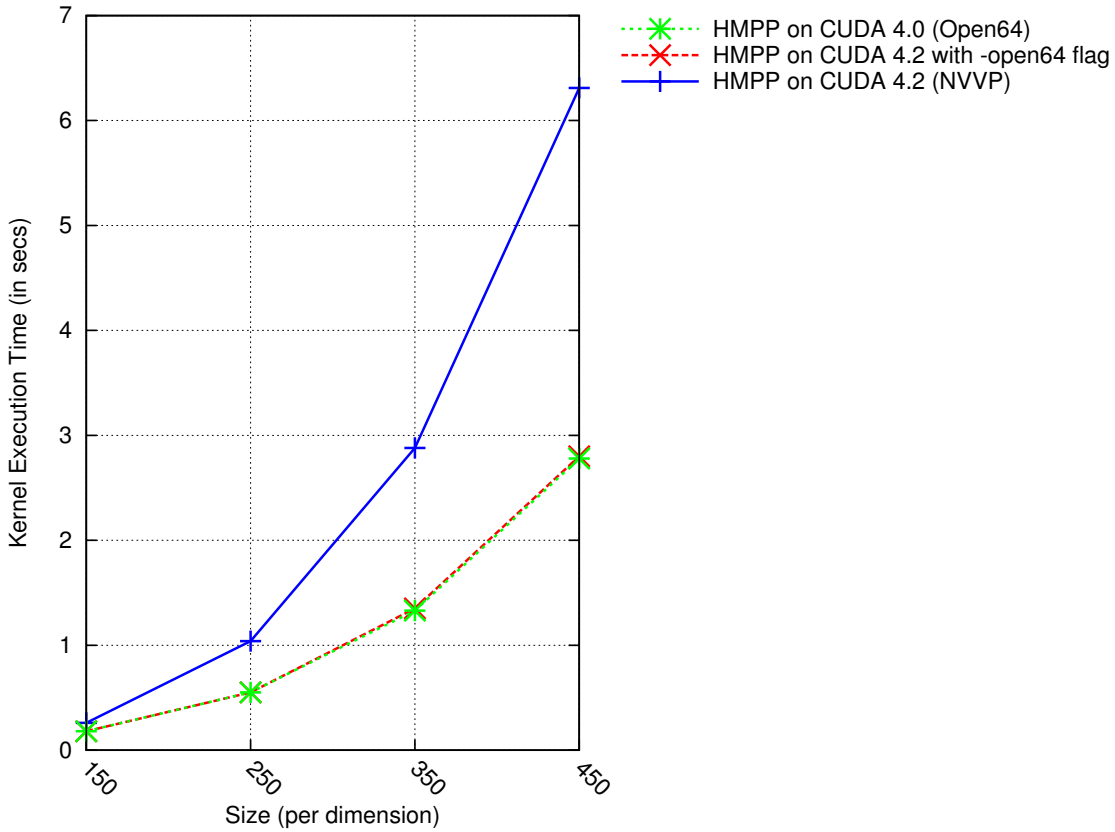
Different compiler front-ends of CUDA - NVVM vs Open64

- Prior to CUDA 4.1, NVCC used Open64 front-end; now LLVM front-end, NVVM
- The -open64 flag instructs NVCC to use Open64 front-end

Different compiler front-ends of CUDA - NVVM vs Open64 (ISO)



Different compiler front-ends of CUDA - NVVM vs Open64 (TTI)



OpenMP blocking implementation on PGI and Intel compilers

- Blocking the two outermost loops (j and k)

OpenMP blocking implementation on PGI and Intel compilers

- Blocking the two outermost loops (j and k)
- Block sizes are different for different data sizes

OpenMP blocking implementation on PGI and Intel compilers

- Blocking the two outermost loops (j and k)
- Block sizes are different for different data sizes
- Collapsing the outer stripped loops (j and k) could yield better performance

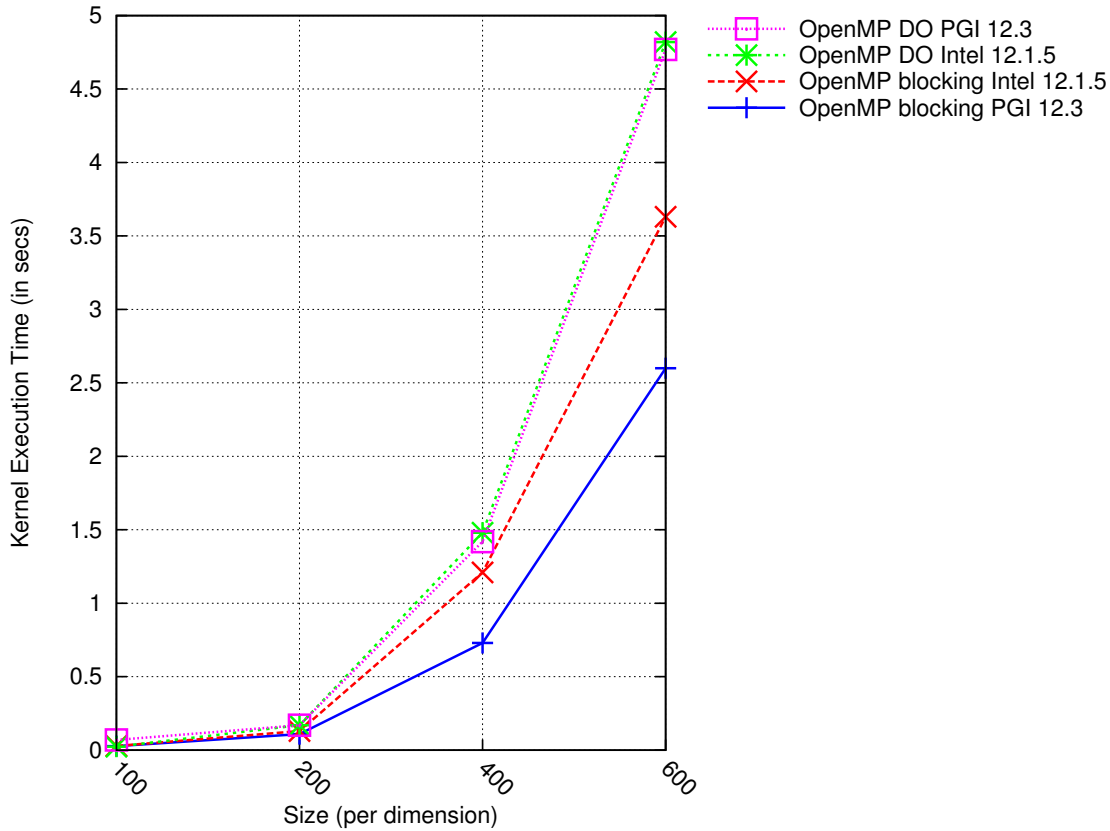
OpenMP blocking implementation on PGI and Intel compilers

- Blocking the two outermost loops (j and k)
- Block sizes are different for different data sizes
- Collapsing the outer stripped loops (j and k) could yield better performance

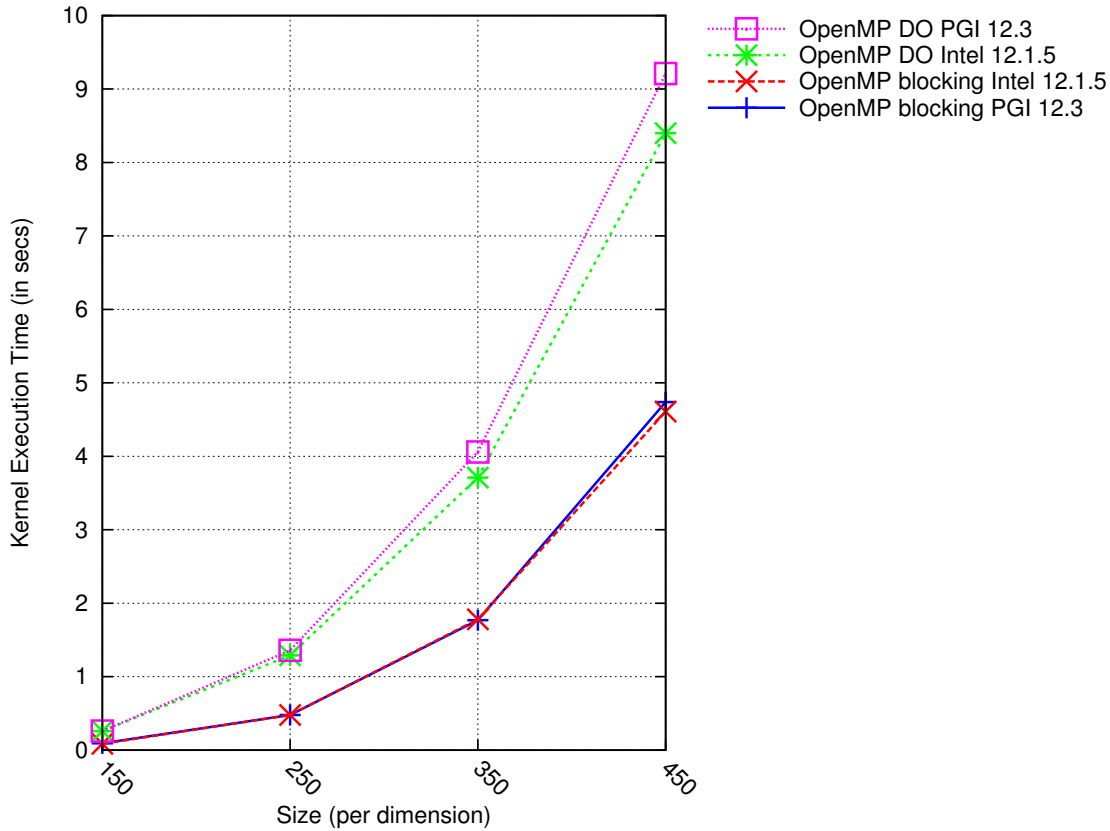
Block sizes for OpenMP cache blocking - ISO and TTI

ISO Data Size	**ISO Block Size(jXk)**	**TTI Data Size**	**TTI Block Size(jXk)**
100	4X4(Intel) / 8X8 (PGI)	150	4X4
200	8X8(Intel) / 16X16(PGI)	250	8X8
400	64X64	350	16X16
600	64X64	450	16X16

OpenMP blocking implementation of ISO on PGI and Intel compiler



OpenMP blocking implementation of TTI on PGI and Intel compiler



Re-evaluation

- **Programmability** - Ease of use, features available, adherence of compilers to specifications.

Re-evaluation

- **Programmability** - Ease of use, features available, adherence of compilers to specifications.
- **Adaptability** - How much of base code change is required to incorporate the directives?

Re-evaluation

- **Programmability** - Ease of use, features available, adherence of compilers to specifications.
- **Adaptability** - How much of base code change is required to incorporate the directives?
- **Performance** - How is the performance w.r.t multithreaded CPU code? Does the compilers provide hints so that performance limiters could be identified?

Criticism

- **Programmability**

Criticism

- **Programmability**
  - Pointers may not work in compute region

Criticism

- **Programmability**
  - Pointers may not work in compute region
  - Implicit model (just !$acc kernels) might not yield the best results

Criticism

- **Programmability**
  - Pointers may not work in compute region
  - Implicit model (just !$acc kernels) might not yield the best results
  - Unsupported/Unstable clauses (`cache`, `async`)

Criticism

- **Programmability**
  - Pointers may not work in compute region
  - Implicit model (just !$acc kernels) might not yield the best results
  - Unsupported/Unstable clauses (`cache`, `async`)
- **Adaptability**

Criticism

- **Programmability**
  - Pointers may not work in compute region
  - Implicit model (just !$acc kernels) might not yield the best results
  - Unsupported/Unstable clauses (`cache`, `async`)
- **Adaptability**
  - Strong chances of code modification if pointers are used (PGI)

Criticism

- **Programmability**
  - Pointers may not work in compute region
  - Implicit model (just !$acc kernels) might not yield the best results
  - Unsupported/Unstable clauses (`cache`, `async`)
- **Adaptability**
  - Strong chances of code modification if pointers are used (PGI)
  - Unsupported language features (Fortran async I/O)

Criticism

- **Programmability**
  - Pointers may not work in compute region
  - Implicit model (just !$acc kernels) might not yield the best results
  - Unsupported/Unstable clauses (`cache`, `async`)
- **Adaptability**
  - Strong chances of code modification if pointers are used (PGI)
  - Unsupported language features (Fortran async I/O)
  - Compilers might not strictly adhere to specifications (!$acc gang in PGI/CAPS)

Criticism

- **Programmability**
  - Pointers may not work in compute region
  - Implicit model (just !$acc kernels) might not yield the best results
  - Unsupported/Unstable clauses (`cache`, `async`)
- **Adaptability**
  - Strong chances of code modification if pointers are used (PGI)
  - Unsupported language features (Fortran async I/O)
  - Compilers might not strictly adhere to specifications (!$acc gang in PGI/CAPS)
- **Performance**

Criticism

- **Programmability**
  - Pointers may not work in compute region
  - Implicit model (just !$acc kernels) might not yield the best results
  - Unsupported/Unstable clauses (`cache`, `async`)
- **Adaptability**
  - Strong chances of code modification if pointers are used (PGI)
  - Unsupported language features (Fortran async I/O)
  - Compilers might not strictly adhere to specifications (!$acc gang in PGI/CAPS)
- **Performance**
  - Compiler front-ends/Compiler options might significantly affect performance (`-nofma`, `-fastmath`)

Criticism

- **Programmability**
  - Pointers may not work in compute region
  - Implicit model (just !$acc kernels) might not yield the best results
  - Unsupported/Unstable clauses (`cache`, `async`)
- **Adaptability**
  - Strong chances of code modification if pointers are used (PGI)
  - Unsupported language features (Fortran async I/O)
  - Compilers might not strictly adhere to specifications (!$acc gang in PGI/CAPS)
- **Performance**
  - Compiler front-ends/Compiler options might significantly affect performance (`-nofma`, `-fastmath`)
  - Asynchronous clause was found to degrade performance

Criticism

- **Programmability**
  - Pointers may not work in compute region
  - Implicit model (just !$acc kernels) might not yield the best results
  - Unsupported/Unstable clauses (`cache`, `async`)
- **Adaptability**
  - Strong chances of code modification if pointers are used (PGI)
  - Unsupported language features (Fortran async I/O)
  - Compilers might not strictly adhere to specifications (!$acc gang in PGI/CAPS)
- **Performance**
  - Compiler front-ends/Compiler options might significantly affect performance (`-nofma`, `-fastmath`)
  - Asynchronous clause was found to degrade performance
  - `cache` clause might cause overheads as compiler might try to use shared memory anyway

Conclusion

- Many options to optimize and port application to a GPU, using only a handful of directives

Conclusion

- Many options to optimize and port application to a GPU, using only a handful of directives
- Underlying implementations of the accelerator models are different (generated CUDA files of PGI and CAPS)

Conclusion

- Many options to optimize and port application to a GPU, using only a handful of directives
- Underlying implementations of the accelerator models are different (generated CUDA files of PGI and CAPS)
- Since compiler does a lot of work, certain switches/options could affect the performance reasonably (e.g `nofma`, compiler front-ends, fast-math, etc)

Conclusion

- Many options to optimize and port application to a GPU, using only a handful of directives
- Underlying implementations of the accelerator models are different (generated CUDA files of PGI and CAPS)
- Since compiler does a lot of work, certain switches/options could affect the performance reasonably (e.g `nofma`, compiler front-ends, fast-math, etc)
- In *effort vs obtained performance* metric, OpenACC is ahead of others (1.8x speedup w.r.t OpenMP version, using almost same number of statements as the OpenMP version)

Conclusion

- Many options to optimize and port application to a GPU, using only a handful of directives
- Underlying implementations of the accelerator models are different (generated CUDA files of PGI and CAPS)
- Since compiler does a lot of work, certain switches/options could affect the performance reasonably (e.g `nofma`, compiler front-ends, fast-math, etc)
- In *effort vs obtained performance* metric, OpenACC is ahead of others (1.8x speedup w.r.t OpenMP version, using almost same number of statements as the OpenMP version)
- Compilers have nice interfaces to profile code

Conclusion

- Many options to optimize and port application to a GPU, using only a handful of directives
- Underlying implementations of the accelerator models are different (generated CUDA files of PGI and CAPS)
- Since compiler does a lot of work, certain switches/options could affect the performance reasonably (e.g `nofma`, compiler front-ends, fast-math, etc)
- In *effort vs obtained performance* metric, OpenACC is ahead of others (1.8x speedup w.r.t OpenMP version, using almost same number of statements as the OpenMP version)
- Compilers have nice interfaces to profile code
- GPU shared memory and asynchronous transfers could siginificantly affect performance

Conclusion

- Many options to optimize and port application to a GPU, using only a handful of directives
- Underlying implementations of the accelerator models are different (generated CUDA files of PGI and CAPS)
- Since compiler does a lot of work, certain switches/options could affect the performance reasonably (e.g `nofma`, compiler front-ends, fast-math, etc)
- In *effort vs obtained performance* metric, OpenACC is ahead of others (1.8x speedup w.r.t OpenMP version, using almost same number of statements as the OpenMP version)
- Compilers have nice interfaces to profile code
- GPU shared memory and asynchronous transfers could siginificantly affect performance
- Compilers should support all language features (like pointers)

Conclusion

- Many options to optimize and port application to a GPU, using only a handful of directives
- Underlying implementations of the accelerator models are different (generated CUDA files of PGI and CAPS)
- Since compiler does a lot of work, certain switches/options could affect the performance reasonably (e.g `nofma`, compiler front-ends, fast-math, etc)
- In *effort vs obtained performance* metric, OpenACC is ahead of others (1.8x speedup w.r.t OpenMP version, using almost same number of statements as the OpenMP version)
- Compilers have nice interfaces to profile code
- GPU shared memory and asynchronous transfers could siginificantly affect performance
- Compilers should support all language features (like pointers)
- Code with too many branches might need to be simplified

Future work

- Using latest compilers/toolkit (PGI, CAPS, Nvidia)

Future work

- Using latest compilers/toolkit (PGI, CAPS, Nvidia)
- Latest GPU (Nvidia GK-110)

Future work

- Using latest compilers/toolkit (PGI, CAPS, Nvidia)
- Latest GPU (Nvidia GK-110)
- OpenMP 4.0 RC1 - SIMD constructs

Future work

- Using latest compilers/toolkit (PGI, CAPS, Nvidia)
- Latest GPU (Nvidia GK-110)
- OpenMP 4.0 RC1 - SIMD constructs
- Target multiple GPUs using Accelerator directives

Acknowledgements

- Georges-Emmanuel Moulard (CAPS enterprise)
- Matthew Colgrove (PGI)
- Philippe Thierry (Intel)
- My colleagues at HPCTools lab (University of Houston)

Accelerator Directives / Kernels	ISO	TTI
CAPS HMPP	1.29	1.76
PGI Accelerator	0.85	1.39
OpenACC	1.54	1.73

Questions?