Heterogeneous parallel algorithms

Combining multicore CPU and GPU programming - the Fast Multipole Method

Marcus Holm
Motivation

Suppose you have a complicated compute-heavy algorithm. Your options are:

- Write a short code for the CPU
  - But then it's limited by the CPU's low FLOPs
- Write a really fast code for the GPU
  - But then it's going to be brutishly complex
- Or use a hybrid approach
Overview

- FMM step by step
  - Algorithm
  - Parallelization
    - CPU
    - GPU
- Performance analysis
- Autotuning
Fast Multipole Method

- Greengard & Rokhlin (1988)
Algorithm overview

Observation: Groups of particles that are far from each other interact only weakly

- So: recursively divide particles into groups
- Identify which groups are weakly coupled
- Perform weak interactions using multipole expansions
- Perform strong interactions directly
Algorithm overview

- **Tree construction**
  - Partition the particles into boxes and sub-boxes
  - Connect the boxes (create the connectivity matrix)

- Initialize bottom of tree
- Sweep upwards
- Sweep down
- Perform far-field interactions on particles
- Perform near-field interactions
Tree construction (1)

- Partition the particles

  for each level in tree
  partition along median of each box

  for each level in tree until enough
  partition along median of each box
  for each box
  **task:** partition subtree
Tree construction (2)

- Calculate box connectivity

for each level in tree
  for each parent on level
    for each child of parent
      children of near connections of parent become either
      near connections or
      far connections of child
\( \theta \)-criterion

- Determine coupling between boxes

\[ R + \theta r \leq \theta d, \]
where
\[ R = \text{size of big box,} \]
\[ r = \text{size of small} \]
\[ d = \text{separation} \]
and \( \theta \in (0, 1) \)
Controlling error

- Approximation error
  - Depends on $\theta$
  - and number of terms in expansions $p$
- We can specify a relative tolerance $TOL$ and have:
  $$p \sim \log TOL / \log \theta$$
Particle to Multipole (P2M)

- Initialize the finest level of tree
- Data moves from particles to leaf nodes

for each leaf box
  calculate multipole expansion

for each chunk of leaf boxes
  task: do P2M as above
Multipole to Multipole (M2M)

- Sweep up the tree
- Data sifts from leaf nodes to root

for each level from leaves to root
shift and add multipole expansions
of children into parent
Multipole to Local (M2L)  
Local to Local (L2L)

- Sweep down the tree
- Data sifts from root to leaf nodes

```
for each level from root to leaves
  for each box on level
    shift M from interaction list to L
    shift L from parent to children
```

```
for each level in tree until enough
  shift M and L
for each box
  task: do ML2L as above
```
Local to Particle (L2P)

- Calculate interactions between particles and the local expansions
- Data moves from leaf nodes to particles

for each leaf box
  for each particle
    calculate interaction with L

for each chunk of leaf boxes
  task: do L2P as above
Particle to Particle (P2P)

- Calculate pair-wise interactions between particles in strongly coupled boxes

for each leaf box
  for each strongly coupled box
    calculate pair-wise interactions

for each chunk of leaf boxes
**task:** do P2P as above
CPU Parallelization with Tasks

![Graph showing speedup vs. threads for different tasks]
Hybrid Parallelization with Tasks
N-body problem on GPU

for each leaf box

*copy p, q to device memory
<on every thread>*

\[ tp = p[\text{threadID}] \]
\[ tU = &U[\text{threadID}] \]

for \( j=1:\text{numBlocks} \)

*load q into shared*

for \( k=1:\text{blockSize} \)

\[ tU += f(tp, \text{shared } q[k]) \]

*copy tU to host memory*
Executing P2P on GPU

- 9.5x faster with 1 CPU thread
- 4.2x faster with full CPU
  - 6x with log potential
Algorithm parameters

- **Distance criterion θ**
  - Small θ: low $p$, large strongly coupled regions
  - Large θ: high $p$, small strongly coupled regions

- $N_{\text{levels}}$
  - Small $N_{\text{levels}}$: small tree, large strongly coupled regions
  - Large $N_{\text{levels}}$: large tree, small strongly coupled regions
Varying $\theta$
Goals of autotuning

- Automatic optimal performance with some conditions:
  - generality (applicability to many problems and situations)
  - robustness (overcoming pathological conditions)
  - speed (finding optimum quickly)
  - efficiency (minimizing any additional computational work)
Autotuning pitfalls

- Highly variable iteration runtimes
  - Noise
  - Quickly changing conditions simulation
- Multiple local optima
- Discontinuous movement of the optimum
- Correlated controls
Autotuning schemes

- Basic idea:
  - Periodically vary $\theta$ or $N_{\text{levels}}$ and see if runtime improves

- Questions:
  - how much (step size)
  - how often
  - in which direction ($\theta$ or $N_{\text{levels}}$, + or -)
Autotuning scheme 1

Algorithm 1: AT1: Random walk

if $\text{time}_i > \text{time}_{i-1}$ then
    return $p_{i+1} = p_{i-1}$
end if

if time to move in levels then
    return $p_{i+1} = [\theta, N_{\text{levels}} + (\text{randint} \times 2 - 1)]$
else if time to move in $\theta$ then
    return $p_{i+1} = [\theta + (\text{randint} \times 2 - 1) \times \text{thetastep}, N_{\text{levels}}]$
else
    return $p_{i+1} = p_i$
end if
Drawbacks of scheme 1

- Has no memory
- Gets stuck in local minima
Algorithm 2: AT2: Directed walk with W-cycle

\[
\text{if } \text{time}_i > \text{time}_{i-1} \text{ then} \\
\quad \text{if previous move was a } \theta\text{-move then} \\
\qquad \text{if fibcount < fiblength then} \\
\quad\quad \text{set thetastep = fib(fibcount + +)} \\
\quad\else \\
\quad\quad \text{set thetastep = 1, fiblength ++} \\
\quad\end{if} \\
\text{end if} \\
\text{end if} \\
\text{set move = } -1 \times \text{move} \\
\text{return } p_{i+1} = p_{i-1} \\
\text{end if} \\
\text{if time to move in levels then} \\
\quad \text{set movedir = [0, 1]} \\
\text{else if time to move in theta then} \\
\quad \text{set movedir = thetastep } \times [1, 0] \\
\text{else} \\
\quad \text{set movedir = [0, 0]} \\
\text{end if} \\
\text{return } p_{i+1} = p_i + \text{move } \cdot \text{movedir}
\]
Drawbacks of scheme 2

- Tuning $N_{\text{levels}}$ is potentially costly

- Two options:
  - Use load balance to inform tuner
  - Place limit on expected tuning cost
Algorithm 3: AT3a: Loadbalance-aware directed walk with W-cycle

if $time_i > time_{i-1}$ then
    if previous move was a $\theta$-move then
        if $\text{fibcount} < \text{fiblength}$ then
            set $\text{thetastep} = \text{fib}(\text{fibcount}++)$
        else
            set $\text{thetastep} = 1, \text{fiblength}++$
        end if
    end if
    set $\text{move} = -1 \times \text{move}$
    return $p_{i+1} = p_{i-1}$
end if

if time to move in levels then
    if CPU waits on GPU then
        set $\text{movedir} = [0, 1]$
    else
        set $\text{movedir} = [0, -1]$
    end if
    return $p_{i+1} = p_i + \text{movedir}$
else if time to move in theta then
    set $\text{movedir} = \text{thetastep} \times [1, 0]$
else
    set $\text{movedir} = [0, 0]$
end if
return $p_{i+1} = p_i + \text{move} \cdot \text{movedir}$
Drawbacks of scheme 3a

- Doesn't apply for CPU-only code
- Doesn't work at all for small problems
- Still potentially costly
Algorithm 4: AT3b: Directed walk with W-cycle and cost estimation

if \( \text{time}_i > \text{time}_{i-1} \) then
  if previous move was a \( \theta \)-move then
    if \( \text{fibcount} < \text{fiblength} \) then
      set \( \text{thetastep} = \text{fib}(\text{fibcount} + 1) \)
    else
      set \( \text{thetastep} = 1, \text{fiblength} + 1 \)
  end if
  set \( \text{thetamove} = -1 \times \text{thetamove} \)
else
  \( \text{cost} = \text{time}_i - \text{time}_{i-1} \)
  if \( \text{movedir} > 0 \) then
    set \( \text{upcost} = \text{upcost} + \text{cost} \)
    set \( \text{upintervall} = \text{basetime} - (\text{upcost} + \text{cost})/\text{cap} \)
  else
    set \( \text{downcost} = \text{downcost} + \text{cost} \)
    set \( \text{downintervall} = \text{basetime} - (\text{downcost} + \text{cost})/\text{cap} \)
  end if
end if
return \( \text{move}_{i+1} = \text{move}_{i-1} \)
else
  set \( \text{basetime} = \text{basetime} + \text{time}_i \)
end if
if time to move in levels then
  set \( \text{movedir} = [0, \pm 1] \) as appropriate
else if time to move in \( \theta \) then
  set \( \text{movedir} = \text{thetastep} \times [\text{thetamove}, 0] \)
else
  set \( \text{movedir} = [0, 0] \)
end if
return \( p_{i+1} = p_i + \text{movedir} \)
Setting max tuning cost

![Graph showing runtime (s) vs tuning cost]

- X-axis: Tuning cost
- Y-axis: Runtime (s)

The graph illustrates the relationship between tuning cost and runtime, indicating a decrease in runtime as the tuning cost increases.
Autotuning results
Autotuning results

![Bar chart showing speedup for different autotuning strategies (none, at1, at2, at3a, at3b) for Small and Large categories.](chart.png)
Worst-case cost
Summary

- Much simpler code than all-GPU code
- Good speedup on 8-core CPU
- >4x faster than CPU-only code
- Autotuning for usability