Execution Strategy and Runtime Support for Regular and Irregular Applications on Emerging Parallel Architectures

Xin Huo
Advisor: Gagan Agrawal
Motivation - Architecture

• Challenges on GPU architecture
  - *Parallel strategies* for different computation patterns
  - Utilize fast and small *shared memory*
  - Efficient and deadlock-free *locking support*
  - Exploit *SIMT* execution manor

• Heterogeneous CPU+GPU architecture
  - Number of CPU+GPU systems increases by fivefold from 2010 to 2012 in top 500 list
  - Emergence of integrated CPU-GPU architecture
    • Fusion APU and Intel Sandy Bridge
  - Task scheduling considering
    • *computation patterns, transmission overhead, command launching overhead, synchronization overhead, load imbalance*
Motivation - Application

• Irregular / Unstructured Reduction
  - A dwarf in Berkeley view on parallel computing (Molecular Dynamics and Euler)
  - Challenges in Parallelism
    • Heavy data dependencies
  - Challenges in Memory Performance
    • Indirect memory accessing results in poor data locality

• Recursive Control Flow
  - Conflicts between SIMD architecture and control dependencies
  - Recursion Support: **SSE (No)**, **OpenCL (No)**, **CUDA(Yes)**

• Thesis Work
  - New software and hardware scheduling frameworks can help map irregular and recursive applications to new architectures.
Thesis Work

• Different strategies for *generalized reductions* on GPU
  - Approaches for Parallelizing Reductions on GPUs (HiPC 2010)

• Strategy and runtime support for *irregular reductions*
  - An Execution Strategies and Optimized Runtime Support for Parallelizing Irregular Reductions on Modern GPUs (ICS 2011)

• Task scheduling frameworks for *heterogeneous architectures*
  - Decoupled GPU + CPU
    • Porting Irregular Reductions on Heterogeneous CPU-GPU Configurations (HiPC 2011)
  - Coupled GPU + CPU
    • Runtime Support for Accelerating Applications on an Integrated CPU-GPU Architecture (SC 2012)

• Improve parallelism of SIMD for *recursion* on GPUs
  - Efficient scheduling of recursive control flow on GPUs (ICS 2013)

• Further extend recursion support
  - Recursion support for vectorization
  - Task scheduling of recursive applications on GPUs
Outline

• Current Work
  - Strategy and runtime support for irregular reductions
  - Improve parallelism of SIMD for recursion on GPUs
  - Different strategies for generalized reductions on GPU
  - Task scheduling frameworks for heterogeneous architectures
    • Decoupled GPU + CPU
    • Coupled GPU + CPU

• Proposed Work
  - Further extend recursion support
    • Recursion support for vectorization
    • Task scheduling of recursive applications on GPUs

• Conclusion
Irregular Reduction

• A dwarf in Berkeley view on parallel computing
  - Unstructured Grid pattern
  - More random and irregular accesses
  - Indirect memory references

• IA
  - Indirection Array
  - Iterates over e (Computation Space)

• RObj
  - Accessed by Indirection Array
  - Reduction Space

/* Outer Sequence Loop */
while()
{
  /* Reduction Loop */
  Foreach(element e)
  {
    (IA(e,0),val1) = Process(IA(e,0));
    (IA(e,1),val2) = Process(IA(e,1));
    RObj = Reduce(RObj(IA(e,0)),val1);
    RObj = Reduce(RObj(IA(e,1)),val2);
  }
  Global Reduction to Combine RObj
Application Context

- Molecular Dynamics
  - Indirection Array -> Edges (Interactions)
  - Reduction Objects -> Molecules (Attributes)
  - Computation Space -> Interactions b/w molecules
  - Reduction Space -> Attributes of Molecules
Main Issues

• Traditional Strategies are not effective
  - Full Replication (Private copy per thread)
    • Large memory overhead
    • Both intra-block and inter-block combination
    • Shared memory usage unlikely
  - Locking Scheme (Private copy per block)
    • Heavy conflicts within a block
    • Avoid intra-block combination, but not inter-block combination
    • Shared memory is only available for small data sets

• Need to choose Partitioning Strategy
  - Make sure data can be put into shared memory
  - Choice of partitioning space (*Computation* VS. *Reduction*)
  - Tradeoffs: Partitioning overhead & Execution efficiency
Contributions

• A Novel Partitioning-based Locking Strategy
  - Efficient shared memory utilization
  - Eliminate both intra and inter-block combination

• Optimized Runtime Support
  - Multi-Dimensional Partitioning Method
  - Reordering & Updating components for correctness and memory performance

• Significant Performance Improvements
  - Exhaustive evaluation
  - Up to 3.3x improvement over traditional strategies
Data structures & Access Pattern

- **Goal**: Utilize shared memory

- **IA**: No reuse, no benefit from shared memory

- **RObj**: Reuse is possible, more benefits from shared memory

```c
/* Outer Sequence Loop */
while()
{
  /* Reduction Loop */
  Foreach(element e)
  {
    (IA(e,0),val1) = Process(IA(e,0));
    (IA(e,1),val2) = Process(IA(e,1));
    RObj = Reduce(RObj(IA(e,0)),val1);
    RObj = Reduce(RObj(IA(e,1)),val2);
  }
  Global Reduction to Combine RObj
}
```
Choice of Partitioning Space

- Two partitioning choices:
  - Computation Space
    - Partition on edges
  - Reduction Space
    - Partition on nodes
Computation Space Partitioning

- Partitioning on the iterations of computation loop

Pros:
- Load Balance on Computation

Cons:
- Unequal reduction size in each partition
- Replicated reduction elements (4 out of 16 nodes are replicated)
- Combination cost

Shared memory is infeasible
Reduction Space Partitioning

- **Pros:**
  - Balanced reduction space
  - Independent between each two partitions
  - Avoid combination cost
  - Shared memory is feasible

- **Cons:**
  - Imbalance on computation space
  - Replicated work caused by the crossing edges
Reduction Space Partitioning - Challenges

• Unbalanced & Replicated Computation
  - Partitioning method can achieve balance between \textit{Cost} and \textit{Efficiency}
    • \textit{Cost}: Execution time of partitioning method
    • \textit{Efficiency}: Reduce number of crossing edges (Replicated work)

• Maintain correctness on GPU
  - Reorder reduction space
  - Update/Reorder computation space
Runtime Partitioning Approaches

• Metis Partitioning (Multi-level k-way Partitioning)  \( \text{High Cost} \)
  - Execute sequentially on CPU
  - Minimizes crossing edges
  - **Cons**: Large overhead for data initialization

• GPU-based (Trivial) Partitioning  \( \text{Low Efficiency} \)
  - Parallel execution on GPU
  - Minimize execution time
  - **Cons**: Large number of crossing edges among partitions

• Multi-dimensional Partitioning (Coordinate Information)
  - Execute sequentially on CPU
  - Balance between *cost* and *efficiency*
Experiment Evaluation

• Platform
  - NVIDIA Tesla C2050 “Fermi” (14x32=448 cores)
  - 2.86 GB device memory
  - 64 KB configurable shared memory
    • 48 KB shared memory and 16 KB L1 cache
    • 16 KB shared memory and 48 KB L1 cache
  - Intel 2.27 GHz Quad core Xeon E5520 with 48GB memory

• Applications
  - Euler (Computational Fluid Dynamics)
    • 20K nodes, 120K edges, and 12K faces
  - MD (Molecular Dynamics)
    • 37K molecules, 4.6 Million interactions
Euler - Performance Gains

- **Euler**: Comparison between Partitioning-based Locking (PBL), Locking, Full Replication, and Sequential CPU time
Molecular Dynamics - Performance Gains

- **Molecular Dynamics:** Comparison between Partitioning-based Locking (PBL), Locking, Full Replication, and Sequential CPU time.

![Graph showing execution time comparison between PBL, Locking, Full Replication, and CPU.]

Execution time (sec)

- PBL: 17.6
- Locking: 5.7
- Full Replication: 2.1
- CPU: 1

Monday, April 8, 13
Comparison of Different Partitioning Schemes - Cost

- **Euler**: Compare Metis Partitioner (MP), GPU Partitioner (GP), and Multi-dimensional Partitioner (MD) on 14, 28 and 42 partitions
- Shows only Partitioning Time - (Init Time + Running Time + Reordering Time)

**Graph Description**:
- **Init Time**
  - MP: largest
  - MD: no initialization
- **Running Time**
  - GP: shortest
  - MD: similar to MP
- **Reordering Time**
  - Similar on three strategies
Comparison of Different Partitioning Schemes - Efficiency

- **Euler:** Compare Metis Partitioner (MP), GPU Partitioner (GP), and Multi-dimensional Partitioner (MD) on 14, 28 and 42 partitions
- Shows **workload** - (include redundant workload caused by crossing edges)

**GP:** involve the most replicated workload and load imbalance

**MD is very close to MP**

---

Monday, April 8, 13
End-to-End Execution Time with Different Partitioners

- **Euler**: End-to-End execution time for Multi-dimensional Partitioner (MD), GPU Partitioner (GP), and Metis Partitioner (MP) on 28 partitions.

- **MP**
  - Partitioning time is even larger than computation time.

- **GP**
  - Too much redundant work slow down the execution.

The PBL scheme with different partitioner on 28 partitions.
Summary

• Systematic study to parallelize irregular reductions on modern GPUs

• A Partitioning-based Locking Scheme
  - on reduction space

• Optimized runtime support
  - Three partitioning schemes
  - Reordering and updating components

• Multi-Dimensional Partitioning can balance cost and efficiency

• Achieve significant performance improvement over traditional methods
Outline

• Current Work
  - Strategy and runtime support for irregular reductions
  - Improve parallelism of SIMD for recursion on GPUs
  - Different strategies for generalized reductions on GPU
  - Task scheduling frameworks for heterogeneous architectures
    • Decoupled GPU + CPU
    • Coupled GPU + CPU

• Proposed Work
  - Further extend recursion support
    • Recursion support for vectorization
    • Task scheduling of recursive applications on GPUs

• Conclusion
Limited Recursion Support on Modern GPUs

- No Support on OpenCL and AMD GPUs
- NVIDIA support recursion from computing capability 2.0 and SDK 3.1
- How is performance?

Focus on intra-warp thread scheduling, and each thread in warp executes a recursive computation
A recursion example on NVIDIA GPU

Execution time is bounded by largest task
Threads Re-convergence in General Branch

/** General Branch */
Fib(n)
{
  if((A || B) && C)
  {
    .......
  }
  else {
    .......
  }
}

Post-dominator is the node through which all paths (across different branches) must pass through.

Immediate post-dominator is the post-dominator which is not dominated by any other post-dominators.
Immediate Post-dominator Re-convergence in Recursion

/* General Branch */
Fib(n)
{
    if(n < 2)
    {
        return 1;
    }
    else {
        x = Fib(n-1);
        y = Fib(n-2);
        return x+y;
    }
}

- Re-convergence can only happen on the same recursion level
- Threads with short branch cannot return until the threads with long branch coming back to the re-convergence point

Monday, April 8, 13
Re-convergence Methods

• Immediate Post-dominator Re-convergence
  - Execution time is bounded by the longest branch
    • For \( N \) threads, each thread executes a recursion task with \( M \) branches

\[
T_{ti} = \text{Execution time of } t\text{th thread on } i\text{th branch}
\]

\[
\text{Total time} = \sum_{i}^{M} \max_{t}^{N} (T_{ti})
\]

• Dynamic Re-convergence
  - Re-convergence can happen \textit{before or after} immediate post-dominator
Dynamic Re-convergence Mechanisms

• Remove the static re-convergence on immediate post-dominator

• Dynamic Re-convergence Implementations
  - Frontier-based Re-convergence
    • Frontier: The group of threads that have the same PC (Program Counter) address with the current active threads
    • Re-convergence is on the same frontier
    • Re-convergence can cross different recursion levels
  - Majority-based Re-convergence
    • Majority group of the threads with the same PC
    • Tend to maximize IPC (Instructions Per Cycle)
    • The threads in minority have chance to be in majority in future
  - Other Dynamic Re-convergence mechanisms
    • Frontier-ret: Schedule return instruction first
    • Majority-threshold: Prevent starvation of threads in minority group
Dynamic Re-convergence Mechanisms

Entry

BB1
bra if(n < 2)

Exit

BB2
bra else

Entry

BB1
bra if(n < 2)

Exit

BB2
bra else

Entry

BB1
bra if(n < 2)

Exit

BB2
bra else

Entry

BB1
bra if(n < 2)

Exit

BB2
bra else
Implementation of Dynamic Re-convergence

• GPGPU-sim simulator
  - A cycle-level GPU performance simulator for general purpose computation on GPUs
  - High simulation accuracy (98.3% for GT200, 97.3% for Fermi)
  - Model *Fermi* micro-architecture

• Stack-based Re-convergence mechanism
  - Stack structure
    • *PC*: Address of the future scheduled instruction
    • *Active Mask*: Represent which threads are active for the corresponding PC (Bitset: 1 represent active)
  - Stack Updating Function
    • Update *PC* and *Active Mask* for different implementations
Frontier-based Dynamic Re-convergence

Step 1
Entry 1: PC0
Block: BB1(PC1)
Next PC: PC0
Active Mask: 111

Step 2
Entry 2: PC0
Block: BB2
Next PC: PC2
Active Mask: 111

Step 3
Entry 2: PC0
Block: BB3(PC1)
Next PC: PC0
Active Mask: 111

Step 4
Entry 3: PC0
Block: BB4(PC2)
Next PC: PC4
Active Mask: 100

Step 5
Entry 3: PC0
Block: BB5(PC1)
Next PC: PC0
Active Mask: 011

Step 6
Entry 3: PC0
Block: BB6(PC2)
Next PC: PC4
Active Mask: 111

Step 6 (Re-convergent)
Entry 3: PC0
Block: BB7(PC1)
Next PC: PC4
Active Mask: 111

Step 7
Entry 4: PC0
Block: BB8(PC2)
Next PC: PC4
Active Mask: 111

Monday, April 8, 13
Majority-based Dynamic Re-convergence

Step 1
- Block | Next PC | Active Mask
- Entry1 | PC0 | 111

Step 2
- Block | Next PC | Active Mask
- BB2 | PC2 | 111

Step 3
- Block | Next PC | Active Mask
- Entry2 | PC0 | 111

Step 4
- Block | Next PC | Active Mask
- BB4 | PC2 | 011

Monday, April 8, 13
Experiment Evaluation

• GPGPU-sim Simulator
  - Model Fermi architecture

• Recursive Benchmarks
  - Small number of recursive branches
    • Fibonacci
    • Binomial coefficients
  - Large number of recursive branches
    • Graph coloring
    • NQueens
  - Dependency between branches
    • Tak Function
  - Only one branch
    • Mandelbrot Fractals
# Performance with Increasing Divergence

<table>
<thead>
<tr>
<th>Threads</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread-0</td>
<td>10 8 6 9 7 5 2 4</td>
</tr>
<tr>
<td>Thread-1</td>
<td>10 8 6 9 7 5 2 4</td>
</tr>
<tr>
<td>Thread-2</td>
<td>10 8 6 9 7 5 2 4</td>
</tr>
<tr>
<td>Thread-3</td>
<td>10 8 6 9 7 5 2 4</td>
</tr>
<tr>
<td>Thread-4</td>
<td>10 8 6 9 7 5 2 4</td>
</tr>
<tr>
<td>Thread-5</td>
<td>10 8 6 9 7 5 2 4</td>
</tr>
<tr>
<td>Thread-6</td>
<td>10 8 6 9 7 5 2 4</td>
</tr>
<tr>
<td>Thread-7</td>
<td>10 8 6 9 7 5 2 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Threads</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread-0</td>
<td>10 8 6 9 7 5 2 4</td>
</tr>
<tr>
<td>Thread-1</td>
<td>8 6 9 7 5 2 4 10</td>
</tr>
<tr>
<td>Thread-2</td>
<td>6 9 7 5 2 4 10 8</td>
</tr>
<tr>
<td>Thread-3</td>
<td>9 7 5 2 4 10 8 6</td>
</tr>
<tr>
<td>Thread-4</td>
<td>7 5 2 4 10 8 6 9</td>
</tr>
<tr>
<td>Thread-5</td>
<td>5 2 4 10 8 6 9 7</td>
</tr>
<tr>
<td>Thread-6</td>
<td>2 4 10 8 6 9 7 5</td>
</tr>
<tr>
<td>Thread-7</td>
<td>4 10 8 6 9 7 5 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Threads</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread-0</td>
<td>10 8 6 9 7 5 2 4</td>
</tr>
<tr>
<td>Thread-1</td>
<td>6 9 7 5 2 4 10 8</td>
</tr>
<tr>
<td>Thread-2</td>
<td>7 5 2 4 10 8 6 9</td>
</tr>
<tr>
<td>Thread-3</td>
<td>2 4 10 8 6 9 7 5</td>
</tr>
<tr>
<td>Thread-4</td>
<td>10 8 6 9 7 5 2 4</td>
</tr>
<tr>
<td>Thread-5</td>
<td>6 9 7 5 2 4 10 8</td>
</tr>
<tr>
<td>Thread-6</td>
<td>7 5 2 4 10 8 6 9</td>
</tr>
<tr>
<td>Thread-7</td>
<td>2 4 10 8 6 9 7 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Threads</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread-0</td>
<td>10 8 6 9 7 5 2 4</td>
</tr>
<tr>
<td>Thread-1</td>
<td>7 5 2 4 10 8 6 9</td>
</tr>
<tr>
<td>Thread-2</td>
<td>10 8 6 9 7 5 2 4</td>
</tr>
<tr>
<td>Thread-3</td>
<td>7 5 2 4 10 8 6 9</td>
</tr>
<tr>
<td>Thread-4</td>
<td>10 8 6 9 7 5 2 4</td>
</tr>
<tr>
<td>Thread-5</td>
<td>7 5 2 4 10 8 6 9</td>
</tr>
<tr>
<td>Thread-6</td>
<td>10 8 6 9 7 5 2 4</td>
</tr>
<tr>
<td>Thread-7</td>
<td>7 5 2 4 10 8 6 9</td>
</tr>
</tbody>
</table>
Performance with Increasing Divergence

• Fibonacci and Binomial Coefficients
  - Perfect speedup on 0-Offset

- With increasing divergence
  - Post-dom decreases by 2.5, 4, and 5 times
  - Majority only decreases by 1.8, 2.1, and 2.2 times
Performance with Increasing Divergence

Task Rotation with different offsets under increasing of divergence

Task Rotation with different offsets under increasing of divergence

Monday, April 8, 13
Scalability with Warp Width

- Fibonacci and Binomial Coefficients
  - When warp width is 4, all versions have similar IPC

- With increasing warp width
  - Majority has the better scalability than both frontier and post-dom
Scalability with Warp Width

NQueens IPC

Graph Coloring IPC

Tak Function

Mandelbrot IPC

Monday, April 8, 13
Summary

• Current recursion support is limited by static re-convergence method

• Dynamic re-convergence mechanisms
  - Re-convergence before or after immediate post-dominator
  - Two implementations
    • Frontier-based method
    • Majority-based method

• Kepler GPU
  - Dynamic Parallelism blocks the executing kernel when calling a new kernel.
  - It is no related to intra-warp scheduling.
Different Strategies for Generalized Reductions on GPUs

- Tradeoffs between *Full Replication* and *Locking Scheme*

- Hybrid Scheme
  - Balance between *Full Replication* and *Locking Scheme*
  - Introduce an intermediate scheduling layer “group” under thread block
    - *Intra-group*: Locking Scheme
    - *Inter-group*: Full Replications
  - Benefits of Hybrid Scheme varies with group size
    - Reduced memory overhead
    - Better use of shared memory
    - Reduce combination cost & conflicts

- Extensive evaluations on different applications with different parameters
Porting Irregular Reductions on Heterogeneous CPU-GPU Architecture

• A Multi-level Partitioning Framework
  - Parallelize irregular reduction on heterogeneous architecture
    • Coarse-grained Level: Tasks between CPU and GPU
    • Fine-grained Level: Task between thread blocks or threads
    • Both use reduction space partitioning
  - Eliminate device memory limitation on GPU

• Runtime Support Scheme
  - Pipeline Scheme overlap partitioning and computation on GPU
  - Work stealing based scheduling strategy provide load balance and increase the pipelining length

• Significant Performance Improvements
  - Achieve 11% and 22% improvement for Euler and Molecular Dynamics
Accelerating Applications on Integrated CPU-GPU Architectures

• **Thread Block Level Scheduling Framework on Fusion APU**
  - Work Scheduling target -> thread blocks (Not devices)
  - Only launch one kernel in the beginning -> *small command launching overhead*
  - *No synchronization* between devices or thread blocks
  - *Inter and Intra-device load balance* is achieved by fine-grained and factoring scheduling policies

• **Locking-free Implementations**
  - Master-worker Scheduling
  - Token Scheduling

• **Applications with different communication patterns**
  - Stencil Computing (Jacobi): 1.6x
  - Generalized Reduction (K-means): 1.92x
  - Irregular Reduction (Molecular Dynamics): 1.15x
Outline

• Current Work
  - Strategy and runtime support for irregular reductions
  - Improve parallelism of SIMD for recursion on GPUs
  - Different strategies for generalized reductions on GPU
  - Task scheduling frameworks for heterogeneous architectures
    • Decoupled GPU + CPU
    • Coupled GPU + CPU

• Proposed Work
  - Further extend recursion support
    • Recursion support for vectorization
    • Task scheduling of recursive applications on GPUs

• Conclusion
SIMD Extensions

• Supported in popular processors
  - SIMD lane width is increased from 128 bit (SSE) and 256 bit (AVX) to 512 bit (MIC)

• Exhaustively studied in Compiler and Runtime Support
  - Memory alignment
  - Irregular accessing
  - Control flow

• No recursion support
  - Dynamic function calls
  - Extensive divergence
Overall idea

• Stack-based recursion support for SIMD extensions
  - Software function calling stack for each SIMD lane
    • SIMD operations
    • Contiguous Memory Accessing
    • Divergence
  - Re-convergence method for SIMD lanes
    • Same strategies on GPU cannot be ported to SIMD extensions directly
Structure of Stack Frame

- **Stack Frame**
  - PC: case number of recursion
    - Check End Case
    - Branch Case
    - Return Case
  - n[]: input values
  - ret[]: return values
Stack Driver

while Not stack.finish() do
  Stack_Frame &ff = stack.getTop();
  pc = ff.PC;
  sf.PC ++;
  if $pc = 0$ then
    if $ff.n = end_value$ then
      stack.sf[stack.top-2].ret += final_value;
      stack.pop();
    end
  else if $pc <= branch_num$ then
    stack.push(input(ff.n, branch));
  else
    stack.sf[stack.top-2].ret += ff.ret;
    stack.pop();
  end
end

Read current PC, and update it for next case
Check End Case
Branch Case
Return Case
## Memory Support

- **Contiguous Memory Accessing**
  - Stacks in Column Major
  - Structure of Array
Other Supports

• Divergence Support
  - Let all lanes W/R the same stack level (Inserting *ghost* frame for the lanes with short branches)
  - Mask operations (Support in MIC)

• Re-convergence for SIMD lanes
  - Immediate post-dominator re-convergence
  - How to implement efficient dynamic re-convergence?
    • Updating stack in different levels
    • Incontiguous Accessing & Potential Parallelsim
Tree-based recursive scheduling Framework

- Challenges of scheduling recursion
  - Dynamic task creation
  - Load imbalance

- Tree-based scheduling framework
  - Hierarchy structure
    - Block Tree, Warp Tree, Thread Tree, Thread Stack
  - Task stealing in the same level
  - Nodes are divided into public and private
    - Only public nodes are available for stealing (reduce locking overhead)
  - Task Stealing in different levels
    - Locking-based stealing for warps and blocks
    - Task redistribution for threads in the same warp when beyond a threshold
Conclusions

- Different strategies for generalized reductions on GPU
- Strategy and runtime support for irregular reductions
- Task scheduling frameworks for heterogeneous architectures
  - Decoupled GPU + CPU
  - Coupled GPU + CPU
- Improve parallelism of SIMD for recursion on GPUs
- Further extend recursion support
  - Recursion support for vectorization
  - Task scheduling of recursive applications on GPUs
Thanks for your attention!

Q & A