CUDA 3SAT-Solver

Boolean Satisfiability VS. nVIDIA CUDA

Multicore Architectures and Programming

27. 6. 2008 | Blaß Thorsten, Schönfeld Fabian

Das Problem

- SAT: Boolean Satisfiability (Dt.: Erfüllbarkeitsproblem)
  - Input: Boolesche Formel in Konjunktiver Normalform (CNF)
    - *Bsp: (a OR b OR $\overline{c}$) AND (a OR x OR $\overline{y}$ OR ..) AND ...*
    - *Terminologie: Klauseln, Variablen, Literale*
    - *Hier: Beschränkung auf 3SAT (jede Klausel enthält max. 3 Literale)*
      - *Jede Boolesche Formel lässt sich auf 3SAT Format abbilden*
    - *Ist nur eine einzige Klausel unerfüllbar, so ist die komplette Formel unerfüllbar*
  - Output: Lösbar / Unlösbar
    - *Vorzugsweise auch lösende Belegung der Variablen*

Anwendung

- SAT: Boolean Satisfiability (Dt.: Erfüllbarkeitsproblem)
  - Erstes bewiesenes NP-Vollständiges Problem[1]
  - Weitreichende Anwendungsgebiete
    - *Künstliche Intelligenz (Automatisiertes Beweisen)*
    - *Kryptographie (Faktorisierung)*
    - *Verifikation*
    - *Automatisierung des elektronischen Entwurfs*

1) S.A. Cook, The Complexity of Theorem proving procedures, Proceedings, Third Annual ACM Symposium on the Theory of Computing, ACM, New York, 1971, 151-158

SAT Lösungsansätze

- Formel mit N Variablen besitzt 2^N mögliche Belegungen
  - SAT ist NP-Vollständig! (Stichwort: Master Reduction)

- Verschiedene Lösungsansätze
  - Brute Force: durchtesten aller Belegungen
  - Random Walk Algorithmen
    - *Findet Lösungen nur mit best. Wahrscheinlichkeit*
  - Backtracking Algorithmen
    - *Gängigstes Verfahren sequentieller Solver*
  - Divide & Conquer Ansatz
    - *Divide == Implizite Parallelisierung*

Divide & Conquer Algorithmus

- Sukzessives erfüllen der aktuell ersten Klausel *c*
  - Eines der max. drei Literale in *c* muss sich zu TRUE ergeben
    - *Jede Variablenzuweisung verkürzt die Formel*
    - *Jede Formel muss ihre aktuelle Belegung mitführen!*

```
bool 3SAT_Solve( Formel F ) {

    if (size(F) < limit)

          return bruteForceSolve( F );
    else
        Clause c = firstClause( F );    // c = {x,y,z}
        Formel f1 = set( F , c.x=true );
        Formel f2 = set( F , c.x=false , c.y=true );
        Formel f3 = set( F , c.x=false , c.y=false , c.z=true );

        return( 3SAT_Solve(f1) || 3SAT_Solve(f2) || 3SAT_Solve(f3) );

}
```

Divide & Conquer Algorithmus

- Beispiel:
  $(\mathbf{a} \vee \overline{\mathbf{b}} \vee \mathbf{d}) \,\&\, (\overline{\mathbf{b}}) \,\&\, (\overline{\mathbf{b}} \vee \overline{\mathbf{c}} \vee \mathbf{e}) \,\&\, (\mathbf{a} \vee \overline{\mathbf{e}}) \,\&\, (\overline{\mathbf{a}} \vee \mathbf{b} \vee \mathbf{c})$

  Belegung: b = FALSE, liefert:

  → $(\mathbf{a} \vee \mathbf{T} \vee \mathbf{d}) \,\&\, (\mathbf{T}) \,\&\, (\mathbf{T} \vee \overline{\mathbf{c}} \vee \mathbf{e}) \,\&\, (\mathbf{a} \vee \overline{\mathbf{e}}) \,\&\, (\overline{\mathbf{a}} \vee \mathbf{F} \vee \mathbf{c})$

  → $(\mathbf{a} \vee \overline{\mathbf{e}}) \,\&\, (\overline{\mathbf{a}} \vee \mathbf{c})$

- Belegungen die der Algorithmus durchführt:
  - a = TRUE → $(\mathbf{T} \vee \overline{\mathbf{b}} \vee \mathbf{d}) = \mathbf{T}$
  - a = FALSE, b = FALSE → $(\mathbf{F} \vee \mathbf{T} \vee \mathbf{d}) = \mathbf{T}$
  - a = FALSE, b = TRUE, d = TRUE → $(\mathbf{F} \vee \mathbf{F} \vee \mathbf{T}) = \mathbf{T}$

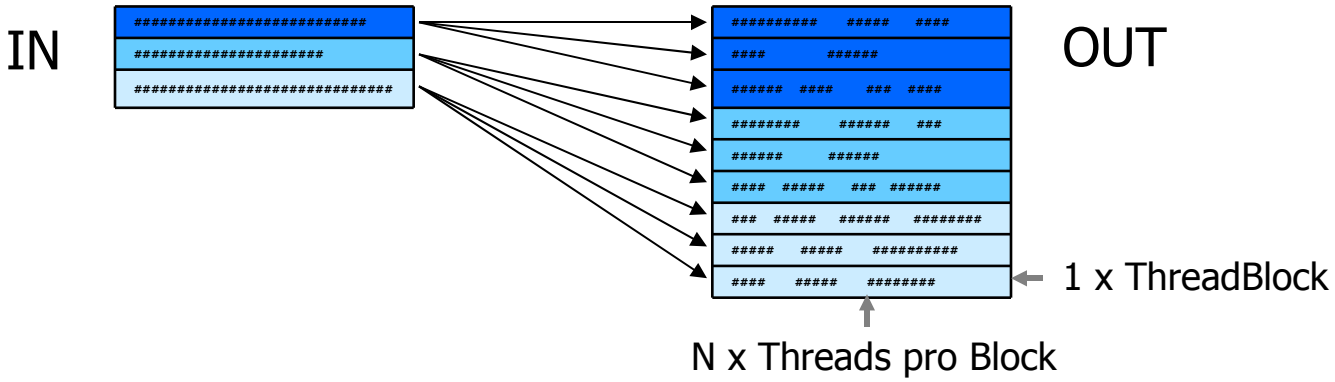
Divide & Conquer Algorithmus

- Prinzip der Parallelisierung
  - Mehrere Formeln gleichzeitig betrachten
  - Alle drei Möglichkeiten der Belegung für jede Formel gleichzeitig durchführen
  - Anwenden von neuen Belegungen Parallel durchführen
    - *Klauseln sind unabhängig voneinander*
  - Kopiervorgänge parallelisieren

CUDA Implementierung

- Parallelisierung auf CUDA Abstraktionen abbilden
  - Ein ThreadBlock übernimmt jeweils eine neue Zuweisung
    - *D.h.: Drei ThreadBlocks haben jeweils die gleiche Eingabe, aber durch versch. Zuweisungen eine andere Ausgabe*
  - Threads innerhalb der Blöcke parallelisieren die Zuweisungen und Kopiervorgänge

- Schematisch:

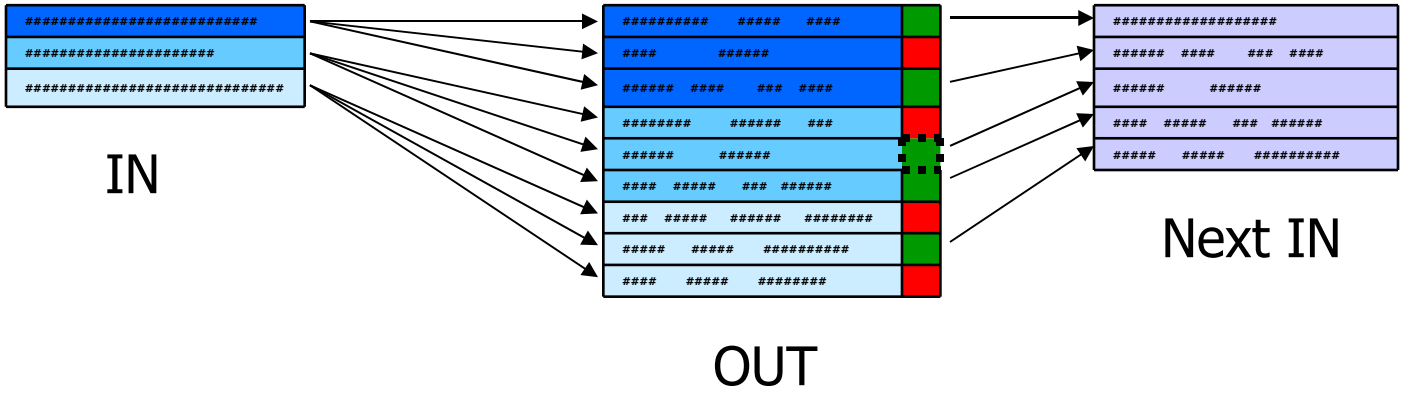


CUDA Implementierung

- Problem: Nicht alle resultierenden Formeln sind gültig
  - Z.b. wird die Klausel (a) durch die Zuweisung a = FALSE sich zu FALSE ergeben und damit die komplette Formel negieren
    - *... ($l_1 v l_2 v l_3$) & (**F**) & ($l_1 v l_2 v l_3$) ... = FALSE*
- Lösung: „einsammeln" der gültigen Formeln und zu neuem Input bündeln
  - Nutzen einer sog. SCAN Operation (Parallel Prefix Scan)
    - *Eingabe: Prädikaten-Array (TRUE/FALSE Werte)*
    - *Ausgabe: Index-Array (Neue Platzierung)*
    - *Schreibt Summe der TRUE Werte bis Index i an Stelle i bzw. i+1*
    - *CUDPP: CUDA Library die versch. Parallele „Building Blocks" implementiert*

CUDA Implementierung

- Zusätzlicher SCAN:



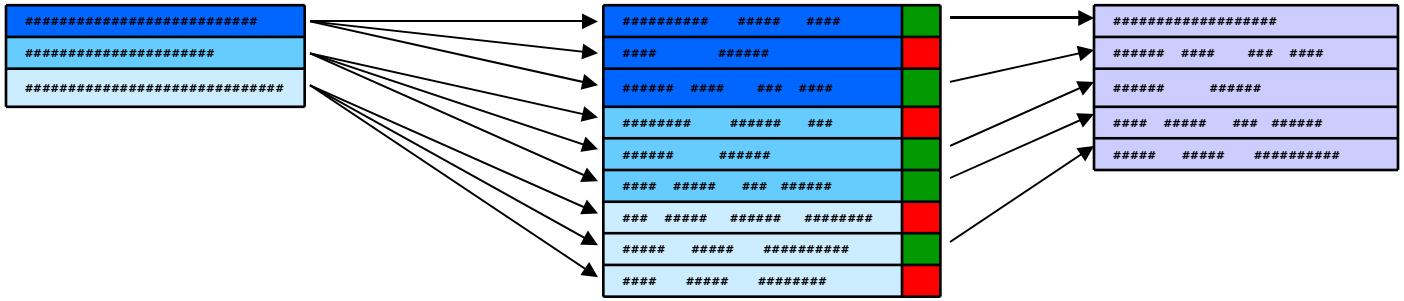
- Parallel Prefix Scan:
  - Prädikatenarray: { 1, 0, 1, 0, 1, 1, 0, 1, 0 }
  - SCAN liefert: { 0, 1, 1, 2, 2, 3, 4, 4, 4 }

Gültige Formel

Neuer Index

CUDA Implementierung

- CUDA Kernels



  - SAT_Kernel
    - *Zuweisen neuer Variablenbelegungen & Verkürzen der Formeln*
    - *Füllen des Predikatenarrays*
  - CUDPP SCAN
  - COPY_Kernel
    - *Gültige Formeln (Predikatenarray) an neue Input-Positionen (SCAN Ergebniss-Array) kopieren*

CUDA Implementierung

- Umgang mit NP-Vollständigkeit
  - Aus jeder Formel können in jedem Schritt bis zu drei neue Formeln entstehen: Speicherbedarf wächst exponentiell
  - Begrenzter Grafikkartenspeicher wird durch Swapping-Mechanik entlastet
    - *Sobald zu viele Formeln im Grafikspeicher liegen, werden Formeln ausgelagert*

CUDA Implementierung

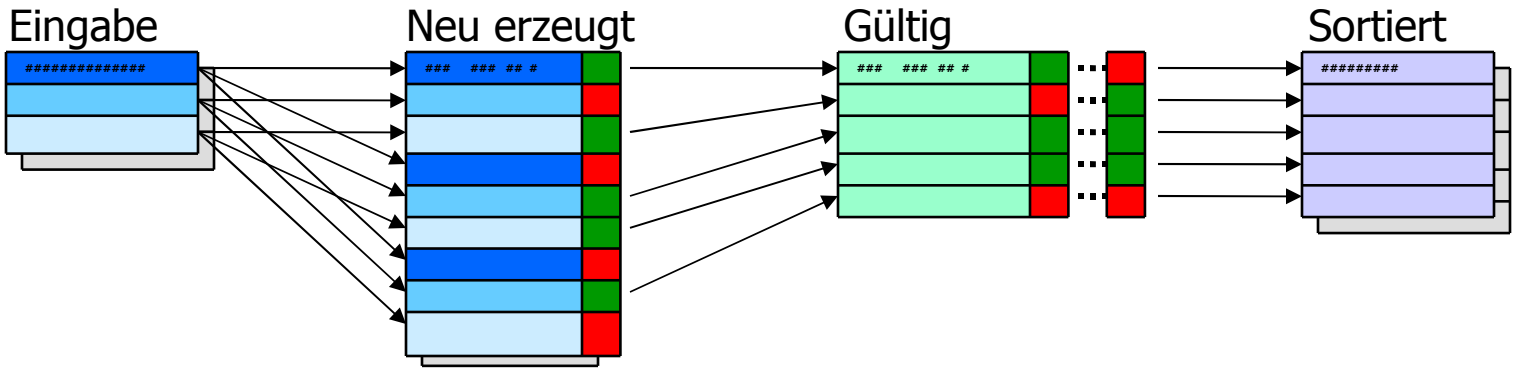
- CUDA Spezifische Anpassungen
  - Shared Memory: Temporäres Auslagern der Formeln zum bearbeiten
    - *Typische CUDA Strategie: Arbeit in Shared Memory laden, bearbeiten, zurück in lokalen Speicher schreiben*
  - Coalescent Memory Access: Formeln werden in ihren Speicherbänken gepolstert (padding) gelagert um Coalescent Access zu ermöglichen
  - Feste Größen im Constant Memory gehalten
    - *Cached Memory*
- Ressourcen Nutzung
  - Derzeit benötigen die Kernels ~ 17 Register
  - Derzeitiger Bottleneck: SAT_Kernel
    - *~60% SAT_Kernel, ~40% COPY_Kernel, memcpy calls weniger als 1%*

CUDA Implementierung

- Erweiterungen
  - Resultierende Zwischenformeln sind fragmentiert
    - *Zusätzlicher SCAN nötig um in den Formeln die noch gültigen Klauseln alle an den Anfang zu bringen*
  - Paralleler D&C Algorithmus implementiert Breitensuche
    - *Abwägen zwischen Tiefen- und Breitensuche*
    - *Gültige Belegungen und ungültige Suchpfade werden erst tiefer im Suchbaum erkannt*
    - *Daher: Pseudo-Sortierung nach erwarteter Länge der neuen Formeln*
      - *Idee: Je mehr Variablen in einem Schritt belegt werden, desto kürzer die resultierende Formel*

CUDA Implementierung

- Komplette Implementierung



- *SAT_Kernel: Variablenzuweisung & Predikatenarray*
- *CUDPP SCAN*
- *COPY_KERNEL: Aufsammeln der gültigen Formeln & Prädikatenmatrix*
- *CUDPP 2D SCAN*
- *DEFRAG_Kernel: Aufsammeln der gültigen Klauseln*

(Grau unterlegt: Aktuelle Variablenzuweisungen pro Formel)

CUDA Implementierung

- Ein Blick in den Code ...

- Beispiel ...

Fragen?

Danke für die Aufmerksamkeit!