Computational Abstraction & Agent-Loop Driven GPU Optimization

Abstract: 본 문서는 계산 모델(Computational Model)의 이론적 추상화부터 시작하여, 이를 현대적인 LLM Agent Loop에 적용하는 방법론을 다룹니다. 특히 RTX 5070(Blackwell) 아키텍처 상에서의 GPU 커널 최적화(MatMul, fMHA)와 문서 자동화 작업을 통합하기 위해, 준정형 DSL(Semiformal DSL) 및 ADT(Algebraic Data Type) 기반의 도메인 모델링 기법을 제안합니다.

1. Computational Model & Abstraction

1.1 개요

계산 모델(Computational Model)은 이론적인 기계에서부터 고수준 시스템 동작에 이르기까지 계산 과정을 추상화한 것입니다. 대수적 도메인 모델(Algebraic Domain Model)은 이러한 추상화를 위한 수학적 프레임워크를 제공하며, 보편 반군(Universal Semigroups), 다항식 링(Polynomial Ring) 표현, 그리고 데이터 구조에 직접 작용하는 관계형 기계(Relational Machines) 등을 통해 계산을 표현합니다.

1.2 핵심 요소 (Key Aspects)

- **정의 (Definition)**: 계산 모델은 상호작용적 행동과 시뮬레이션 기능을 포함하여 계산 과정을 설명하는 수학적 프레임워크입니다.
- **추상화 수준 (Abstraction Levels)**: 물리적 기계의 세부 사항을 숨기고 알고리즘과 데이터에 집중합니다. 기계 중심 언어에서 벗어나 복잡한 상호작용 시스템을 분석하는 "고수준" 모델로 이동합니다.
- **유형 (Types)**: 순차적(Sequential), 함수형(Functional), 병행(Concurrent) 모델 등이 포함됩니다.
- **주요 예시**:
  - Turing machines (Small-scale)
  - Boolean circuits
  - Branching programs (Small-space)

- Relational machines (Database-oriented)

1.3 대수적 도메인 모델 (Algebraic Domain Models)

- **계산의 관점**: 계산은 유한 보편 반군(Finite Universal Semigroup)으로부터의 사상(Morphism, 구조를 보존하는 맵)으로 간주될 수 있습니다.
- **도메인 표현**: 개념(Concepts), 역할(Roles), 데이터 타입, 규칙 등을 통해 지식 도메인을 표현하며, 이는 종종 기술 논리(Description Logic)에 기반합니다.
- **대수적 복잡도**: 다항식 링과 같은 대수적 구조를 사용하여 계산 가정을 분석하며, 유한체(Finite Fields) 위에서 알고리즘을 설계합니다.
- **관계형 기계 (Relational Machines)**: 튜링 머신과 달리, 입력 문자열이 아닌 수학적 구조(예: 데이터베이스의 튜플)에 직접 연산을 수행하여 "순서 불일치(Order Mismatch)" 문제를 극복합니다.

1.4 추상화의 구성 요소 및 핵심 이론

- **Data Abstraction**: 복잡하고 저수준인 데이터 표현을 숨겨 소프트웨어 설계를 단순화합니다.
- **Symbolic Computing**: 대수적 명세와 통합 도메인 프레임워크 내에서 추상 계산 구조(ACS)를 사용합니다.
- **Peter Wegner's Perspective**: 상호작용(Interaction)이 알고리즘보다 더 강력하다고 주장합니다.
- **Joel Spolsky's "Leaky Abstractions"**: 모든 고수준 추상화는 저수준의 세부 사항을 완전히 숨길 수 없지만(Leaky), 복잡성을 관리하는 데 여전히 유용합니다.

2. Agent Loop를 위한 준정형 DSL (Semiformal DSL)

에이전트 루프를 통해 문서 자동화나 GPU 커널 최적화 같은 복잡한 작업을 수행할 때, 핵심은 **"추상적 의도"**를 **"구체적 실행 코드/텍스트"**로 점진적으로 변환하는 가교 역할을 하는 **준정형 DSL(Semiformal DSL)**을 설계하는 것입니다.

2.1 설계 원칙

순성형 DSL은 LLM의 유연성(Natural Language)과 코드의 엄격함(Formal Syntax)을 결합해야 합니다.

1. **상태 가독성**: 사람이 읽기 쉬운 텍스트(Markdown)와 구조화된 데이터(JSON/YAML)가 결합된 형태.
2. **부분 수정 (Partial Edits)**: 전체 코드를 다시 쓰는 대신, 특정 블록이나 속성만 수정할 수 있는 구조.

2.2 DSL 정의 및 활용 단계 (Refinement Loop)

Step 1: 아티팩트 중심의 DSL 스키마 정의

작업의 최종 목표인 '아티팩트'를 먼저 정의합니다. (예: GPU 커널 최적화 계획)

```
kernel_plan:
  strategy: "Tiling"
  shared_memory: "16KB per block"
  optimization_hints:
    - "Reduce bank conflicts in shared memory"
    - "Unroll innermost loop for vectorization"
```

Step 2: Feedback-Driven Refinement 루프 구성

- **Generator Agent**: 초기 DSL 초안 작성.
- **Evaluator Agent/Tool**: 프로파일링 결과나 논리적 오류를 분석하여 DSL 업데이트.
- **Refiner Agent**: 업데이트된 정보를 바탕으로 실제 코드(CUDA/Triton) 수정.

Step 3: 실행 정보 반영 (Runtime Centric)

LDB(Large Language Model Debugger) 방식처럼, 실행 중 발생하는 변수 값이나 흐름 정보를 DSL 컨텍스트로 포함합니다.

2.3 도구 및 프레임워크

- **Grammar-Constrained Decoding**: Formatron 등을 사용하여 문법 준수 강제.

- **Domain-Specific Optimization**: Triton, Halide 등의 기존 추상화 계층을 중간 매개체로 활용.

3. MatMul & fMHA 최적화를 위한 계층적 정책 DSL

MatMul(행렬 곱셈)과 fMHA(Fused Multi-Head Attention)는 데이터 재사용성(Locality)과 메모리 계층 활용이 핵심입니다.

3.1 Semiformal DSL 구조 예시

하드웨어의 물리적 제약(정형)과 최적화 전략의 의도(자연어)를 결합합니다.

```
# Artifact: Kernel_Specification_v1
operation: "fMHA"
precision: "FP16"
hierarchy:
  grid: [Batch, Heads, Seq_M / Tile_M, Seq_N / Tile_N]
  block:
    tile_size: [128, 64] # 정형 데이터
    shared_memory: "Double Buffering applied" # 에이전트의 의도
    sync_policy: "Async copy from Global to Shared"

optimizations:
  - stage: "Global_to_Shared"
    method: "Vectorized_Load"
    constraint: "Alignment 16-byte"
  - stage: "Compute"
    method: "Warp_Group_MMA" # Hopper/Blackwell 타겟팅
    refinement_note: "Check register pressure if tile size increc
```

3.2 점진적 Refinement 루프

1. **Skeleton Generation**: 수학적 정의를 바탕으로 기본 Tiling 구조 초안 작성.
2. **Constraint Propagation**: 공유 메모리, 레지스터 개수 등 하드웨어 제약 주입.
3. **Simulation/Profiling**: Nsight Systems, Triton Profiler 결과를 피드백으로 전달.
4. **Policy Update**: 피드백을 바탕으로 `refinement_note` 수정 및 Codegen 반복.

3.3 해석 DSL 초안화 요소

3.3 핵심 DSL 추상화 요소

- **Tiling Algebra**: (M, N, K) 공간 분할 방식 (Row-Major vs Z-Curve).
- **Memory Movement**: Load -> Compute -> Store 파이프라인 제어.
- **Fusion Logic**: fMHA 전용 (Online Softmax 로직 기술).

4. RTX 5070 (Blackwell) & cuTile 최적화 전략

Blackwell 아키텍처는 향상된 Tensor Core 성능과 **TMA(Tensor Memory Accelerator)**를 특징으로 합니다.

4.1 Blackwell 특화 DSL 설계

```
# Agent-Refinable cuTile Specification
kernel_spec = {
    "target": "RTX_5070_Blackwell",
    "operation": "fMHA_Forward",
    "tiles": {
        "BLOCK_M": 128, # 에이전트가 점진적 선택 (64, 128, 256)
        "BLOCK_N": 64,
        "BLOCK_K": 32
    },
    "stages": {
        "load": "TMA_Async",          # Blackwell 효율적 로드
        "compute": "WMM_FP16_Accum",  # Warp Group MMA
        "epilogue": "Online_Softmax"
    },
    "refinement_state": {
        "current_bottleneck": "Shared_Memory_Bank_Conflict",
        "applied_fix": "Swizzling_Layout_V2"
    }
}
```

4.2 단계별 최적화 (The Loop)

- **Stage 1 (Functional Mapping)**: Naive Tiling DSL 생성 및 매핑.
- **Stage 2 (Resource Constraints)**: RTX 5070의 SM당 Shared Memory/Register 한계 주입 및 조정.
- **Stage 3 (Micro-Architecture Tuning)**: Pipeline Latency 은닉(Multi-buffering), Memory Swizzling 적용.

5. Functional Domain Modeling with ADT

Artifact Generation 과정에서 **ADT(Algebraic Data Type)**를 적용하면 에이전트 루프의 안정성과 성능을 비약적으로 높일 수 있습니다.

5.1 핵심 이점

1. **상태 공간의 엄격한 제어 (Sum & Product Types)**
   - Product Types (AND): 유효한 설정값 조합 정의.
   - Sum Types (OR): 최적화 전략을 엄격히 분리 (e.g., `TMA_Direct | Shared_Memory_Staging`).
   - *효과*: 논리적 모순(Impossible State) 방지.
2. **점진적 변환의 수리적 보장**
   - DSL을 수식(Expression)처럼 취급하여, 변환 과정(MatMul -> Tiled_MatMul)의 동등성(Isomorphism) 보장.
3. **패턴 매칭 (Pattern Matching)**
   - 강력한 Decision Table 역할 수행.

```
-- 준정형 DSL의 내부 표현 예시
data MemoryAccess = Global AccessPattern | Shared BankConflic
data OptimizationAction = ApplySwizzling | IncreaseStage Int
```

4. **컴파일 타임 에러 검증**: 실행 전 하드웨어 제약 조건 위반 여부 확인.
5. **Composability**: 연산 조각을 레고 블록처럼 조합하여 퓨즈드 커널 설계.

6. 통합 실행 모델 (Meta-DSL): Intent to Report

GPU 커널 최적화와 문서 자동화라는 서로 다른 도메인을 **"Intent -> Transform -> Effect -> Report"**라는 동일한 대수적 흐름으로 통합합니다.

6.1 GPU Kernel Optimization DSL

```
# 1. Domain Algebra: 연산 구조
data KernelOp = MatMul(M, N, K) | FMHA(Batch, Head, Seq, D)

# 2. Effect Algebra: 최적화 변환
data OptEffect =
```

```
data OptEffect =
    | Tiling(size_m, size_n, size_k)
    | Pipelining(num_stages)       # Blackwell TMA 활용
    | MemoryLayout(swizzling_bit)  # Bank Conflict 제거

# 3. Executable Model
kernel_plan = {
    "intent": MatMul(M=4096, N=4096, K=4096),
    "invariants": ["result == pytorch_reference", "shared_mem < 1
    "strategy_space": { ... },
    "measurement": ["TFLOPS", "Latency_ms"]
}
```

6.2 Documentation & Report DSL

```
# 1. Domain Algebra: 지식 단위
data KnowledgeSource = MarkdownFile(path) | FactualClaim(source,

# 2. Effect Algebra: 텍스트 변형
data DocEffect =
    | Summarize(word_limit)
    | CrossReference(list_of_claims)
    | SynthesizeThesis(style="Academic")
    | RenderHTML(template="index.html")

# 3. Executable Model
report_plan = {
    "intent": "Blackwell GPU 최적화 실험 보고서 작성",
    "sources": ["./experiment_logs/*.md", "NVIDIA_Whitepaper.pdf'
    "laws": ["Every claim must have a citation"],
    "workflow": [Summarize, ExtractClaims, SynthesizeThesis, Renc
}
```