arXiv:2603.02597v1 [cs.CL] 3 Mar 2026

GPUTOK：GPU 加速的字节级 BPE 分词

Venu Gopal Kadamba KanishkhaJaisankar 数
据科学硕士 数据科学硕士 纽约大学 纽约大学
vk2636@nyu.edu kj2675@nyu.edu

摘要

随着大语言模型的上下文窗口逐渐扩展至百万词元，CPU 分词器成了主要的性能瓶颈，因为它们一次只处理一个步骤，而强大的 GPU 却闲置未用。我们构建了一个基于 GPU 的字节级 BPE 分词器，遵循 GPT-2 的合并规则。它包含一个基础的 BlockBPE 风格内核，以及一个更快的优化版本；后者使用 cuCollections静态映射、CUB 归约，并通过 pybind11 提供 Python 接口。

在 WikiText103 上处理最长达 131k 词元的序列时，优化的GPU分词器生成的词元与 CPU 版本相同；对于最长输入，其速度约为 tiktoken 的 1.7×倍，也约为 HuggingFace GPT-2分词器的 7.6× 倍。Nsight 性能分析显示，CUDA API时间的 70– 80% 用于内存分配，因此，接下来增加内存池应能带来最大的速度提升。使用 WikiText103 提示词进行的生成任务测试表明，我们的 GPU 分词器输出在相似度和重叠度指标上与 tiktoken 和 HuggingFace GPT-2 的差距约在一个百分点以内，这意味着它在保持输出质量的同时，让长上下文推理变得更加实用。

1 引言

大语言模型的上下文窗口正从几千个词元扩展到几十万个，甚至上百万个词元。在这种长上下文场景下，仅加速 Transformer 层还不够：分词等预处理阶段可能占据端到端延迟的很大一部分，即使有强大的 GPU 可用，也会拖慢响应。如果分词仍由单个 CPU 核心完成，GPU 可能会闲置，而 CPU 则缓慢地将原始文本转换为词元，这既浪费硬件资源、增加服务成本，也会限制需要处理长对话、完整文档或文档集合的实时应用。即使是经过充分优化的 CPU 分词器，例如 tiktoken 和 HuggingFace 分词器，也大多采用顺序处理，无法利用现代 GPU 强大的并行性。因此，随着上下文窗口不断增大，GPU 感知型分词器便成为降低延迟的重要组成部分。

字节对编码（BPE） [1] 广泛用于包括 GPT2 [2], 在内的仅解码器大语言模型，但其标准贪心算法是迭代且依赖数据的：它会根据合并表反复查找最佳相邻对，并合并其最左出现，直到无法再进行合并。由于每一轮合并都依赖前一轮、序列长度会在每次合并后发生变化，而且实际词汇表需要进行大量哈希表查找，同时还必须与经过验证的 CPU 实现完全一致，因此很难将这种行为高效映射到 GPU。简单地将 CPU 算法移植到 GPU 上无法获得良好性能；相反，我们的目标是利用

每轮合并中的数据并行性，同时保留与 GPT2 兼容的贪心语义，并基于 BlockBPE 框架 [1] 来指导这一设计。

1.1 我们的方法

我们基于 BlockBPE 框架 [1]设计了一种字节级 BPE 的 GPU实现，兼容 GPT-2 风格的分词，并针对长上下文工作负载进行了优化。总体而言，我们的方法如下：

1. **用于 GPT-2 合并的 GPU 内核**：我们实现了两个遵循 GPT-2 合并优先级的 CUDA 内核：一个基线内核，复现 BlockBPE 的全局最小值贪心合并；另一个优化内核，每轮合并非重叠局部极小值，在保持相同合并顺序的同时减少迭代次数。

2. **简化压缩**：我们采用一种适用于所有序列长度的简单双缓冲压缩方案，可减少共享内存用量并避免复杂的前缀和逻辑，从而使内核更易于调优。

3. **Python 集成**：我们通过基于 pybind11 的 Python 接口封装该实现，重建 GPT2 字节编码器，为合并操作构建驻留于 GPU 的静态映射，并提供适用于基于 PyTorch 的服务流水线的批量分词 API。

第 2 节回顾了 BPE、BlockBPE 以及分词和 GPU 哈希表相关工作。第 3 节详细介绍我们的系统架构和内核设计。第4 节介绍实验设置和基准测试，第 5 节报告实证结果和性能分析。第 7 节总结研究发现，并概述未来工作的方向。

开源实现包含 CUDA 内核、Python 绑定和基准测试脚本，可在此处获取：
https://github.com/venugopalkadamba/gpu-tokenizer。

2 文献综述

现代大语言模型的分词处于子词算法、高性能系统设计和面向 GPU 的数据结构的交汇处。既有研究可分为三大类：（1）字节对编码（BPE）的算法变体；（2）CPU 与 GPU 上的系统和实现策略；（3）支持可扩展分词的 GPU 原语。这种分类不仅揭示了合并操作的定义方式，也阐明了如何在实际服务环境中高效执行这些操作。

BPE 的算法变体。 经典 BPE [3]通过学习频繁出现的成对合并来构建子词词表，而 GPT-2 采用一种字节级变体，将字节映射为可打印的 Unicode [2]，从而避免产生无效 UTF-8 序列。后续研究从更广泛的视角重新审视 BPE：例如，以公平性为导向的目标，如奇偶性感知 BPE [5] ，旨在使不同语言的压缩率趋于一致；而 BPE 丢弃 [4] 等随机推理方法则探索合并应用的鲁棒性与可变性。这些研究揭示了广阔的设计空间，但主要关注应如何选择合并，而非如何在大语言模型服务流水线的资源与延迟约束下执行合并。

系统与实现策略。 从系统角度来看，分词器主要分为三类。基于正则表达式的 CPU 分词器，包括最初的 GPT-2 实现和许多 HuggingFace 实现，会先进行语言感知的预分词，再应用 BPE。它们具备出色的文本处理能力，但正则表达式处理开销较大，且控制流不规则，因此 GPU 加速较为困难。tiktoken 和 HuggingFace Tokenizers 等经过优化的 CPU 库，通过缓存感知型工程、单指令多数据和低开销绑定，使吞吐量接近硬件极限。它们的单节点性能十分出色，但分词期间 GPU 处于空闲状态，导致其与以 GPU 为中心的推理技术栈不匹配。

第三类研究将分词迁移到加速器上。BlockBPE [1] 表明，可以在每轮中评估所有相邻词对，并结合驻留于GPU的哈希表和块级扫描，实现 BPE 的并行化。一些变体放宽了严格的贪心语义，允许每轮执行多个互不重叠的合并，从而大幅提升 GPU 利用率，但代价是生成的词元序列可能偏离 CPU 参考实现。其他面向 GPU 的研究采用字节级简化或近似合并方案，提升了吞吐量，却降低了与 GPT-2 风格词汇表的兼容性。

GPU 哈希表与并行原语。 高效的 GPU 分词依赖驻留于设备的数据结构，以及能够大规模运行的集体操作。cuCollections 提供了高吞吐量的 GPU 哈希映射，例如 static map，非常适合静态合并表。同样，CUDA C++ 核心库提供了 CUB 的块级扫描和归约，这些已成为 GPU 数据并行工作负载中的标准工具。尽管这些组件为 GPU 分词器提供了基础构件，现有文献却未系统研究如何将它们组合成一个与 GPT-2 合并规则兼容、严格遵循贪心语义的 BPE 流水线。

现有方法的优缺点。 CPU 正则表达式分词器擅长处理语言相关行为，并能保证与训练时分词精确兼容，但并行性有限、预处理成本高，且存在 CPU–GPU 数据传输开销。优化后的 CPU 库显著提升了吞吐量，但仍受 CPU 并行性的根本限制，因此无法满足长上下文、GPU 密集型推理的需求。GPU 加速的 BPE 变体展现出显著的加速比和出色的可扩展性，但许多方法放宽了贪心合并语义，或采用简化的预分词，因此不适合作为 GPT-2 风格模型的直接替换实现。

现有工作为何仍有不足。 总的来说，先前的工作在精确的 GPT-2 语义与以 GPU 为中心的系统设计的交汇处留下了空白。算法论文未涉及系统级性能；CPU 库虽能确保正确性，却未充分利用 GPU 资源；而 GPU实现则优先考虑吞吐量，而非精确等价性。此外，尽管 cuCollections 和 CUB 提供了所需的基本组件，但尚无现有工作展示如何将它们结合起来，构建一个符合 GPT-2 合并优先级、提供实用 Python API，并在真实的长上下文工作负载上经过严格评估的分词器。

本研究通过设计一款使用 cuCollections 哈希映射和块级归约的 GPU 常驻 BPE 分词器，探索这一尚未得到充分研究的领域；该分词器在无缝集成到基于 Python 的大语言模型服务流水线的同时，保留了 GPT-2风格贪心合并。其目标是在实现精确兼容性的同时提供高吞吐量，弥补此前以CPU和GPU为中心的方法的不足。

2.1 相对于BlockBPE及早期GPU分词器的定位 s

BlockBPE [1]据我们所知，是首个端到端的 BPE分词 GPU实现。它省去了 2× 基于正则表达式的预分词，使用驻留于GPU的哈希映射和块级前缀扫描，并展示了在高批量工作负载下，吞吐量最高约比`tiktoken`高出 2.5×，比 HuggingFace Tokenizers 高出。在此代价下，BlockBPE 接受其行为与规范 CPU 分词器存在细微差异——最显著的是省去了基于正则表达式的预分词——并明确针对高吞吐量批量推理进行了优化，而非面向长单序列场景。

我们的工作目标更为聚焦，但侧重点有所不同。我们将 GPT-2风格的字节级BPE 视为固定不变的对象，旨在为广泛使用的 CPU 分词器提供严格的直接替换实现：我们重构原始字节编码器，保留贪心合并语义，并要求在所有基准测试中与 CPU参考实现达到令牌级一致性。在系统层面，我们探索了一种特定的设计方案：在每个序列对应的单个 CUDA 块内，结合 cuCollections静态映射与轻量级双缓冲压缩，并将其集成到一个适用于常见基于PyTorch的服务框架的最小化 Python API 中。

我们并不声称在算法层面超越 BlockBPE，也不试图匹配其对批处理场景或调度策略的广泛探索。相反，我们的贡献应被视为一项聚焦的案例研究：探讨如何通过一组少量且明确的工程选择，将严格兼容的 GPT-2分词器迁移到 GPU 上，同时实证分析该设计在何时有所帮助，以及在哪些方面仍存在不足。

3 所提方法

下面介绍整体架构、GPT-2字节编码的重建方式、合并表在 GPU 上的存储方式，以及内核的组织结构。遵循 BlockBPE [1]设置，我们的设计必须同时满足两点：(1) 精确复现 GPT-2 的贪心合并语义；(2) 提供足够细粒度的并行性，使在 GPU 上运行分词器切实可行。下文围绕这两个目标展开：对于每个组件，我们都会说明它的功能，以及为何采用这种组织方式。

3.1 系统架构

总体而言，分词按以下阶段进行：

1. **字节编码（CPU）**。输入的 UTF-8 文本会转换为原始字节，然后通过以 C++ 实现的确定性字节编码器映射为 GPT2风格符号。 2. **分块与批处理（CPU）**。生成的令牌序列可选择拆分为固定大小的数据块（不超过配置的词元预算），使每个数据块都能放入 GPU共享内存，并分配给单个 CUDA 块。 3. **合并遍次（GPU）**。对于每个数据块，一个 CUDA 块会将词元加载到共享内存中，并完全在设备上运行贪心BPE合并循环，使用驻留于GPU的哈希表进行对查找，并通过块级归约求取最小值索引。

4. **组装（CPU）**。所有数据块合并后的词元序列都会被复制回来，按原始顺序拼接，并作为最终的词元 ID 返回。

在我们的实验中，我们选择了合适的序列长度，使每个序列都能由单个块处理，从而简化同步并避免块间通信。对整本书进行性能分析时，我们启用主机端分块，以限制每个块的共享内存占用。我们也考虑过更复杂的方案，即将单个序列分散到多个块中，但这需要额外的全局协调才能保证 GPT-2 的最左贪心语义；相比之下，每序列单块设计使合并顺序确定地局限于块内，因此更易于分析和验证。

3.2 GPT2byte 编码器重建

GPT-2 使用自定义的字节到 Unicode 映射，以避免出现未知词元和无效的 UTF-8 序列。我们在 C 中重新实现了此映射++，并遵循原始参考逻辑。简化版本如清单1所示。

```
结构体 ByteEncoder {
    std::vector<std::string> 字节_转为_符号;            // size 256

    ByteEncoder() {
        // Map printable ranges to themselves (bytes already "safe" as UTF-8)
        for (整数 c = '!'; c <= '~'; ++c) { /* ... */ }
        for (int c = 0xA1; c <= 0xAC; ++c) { /* ... */ }
        for (int c = 0xAE; c <= 0xFF; ++c) { /* ... */ }

        // Map remaining bytes to shifted codepoints
        int n = 0;
        for (整数 字节 = 0;字节 < 256;++字节){
            如果 (/* not already mapped */) {
                字节_转_符号[字节] = 码点_转_UTF-8(256 + n++);
            }
        }
    }
};
```

清单 1：字节到符号的编码（简化版）

在主机上，我们首先应用编码器，生成由 256 种字节级符号组成的序列，然后使用 BPE 词汇表（第2节）将这些符号映射为整数令牌 ID。后续合并操作完全基于整数令牌 ID 进行，因此 GPU 端的所有工作都使用定宽整数完成。重现原始 GPT2 字节编码器，而不是另行设计一种更简单的方案，对于确保兼容性至关重要：这可以保证我们的合并表、字节级表示和最终 ID 与现有的 GPT2 模型及分词器实现保持一致。

3.3 合并表表示

GPT2 合并词汇表存储在 merges.txt 中，包含约五万条合并规则。我们将其解析为两个表，这两个表共享相同的底层整数令牌 ID：

- 一个位于 CPU 上的 std::unordered 映射<PairKey、PairInfo> ，用于参考实现 -化。
- 一个运行于 GPU 上的 cuco::static map<PairKey, PairVal> ，供 CUDA 内核使用。

每对相邻词元都以打包的 64 位形式表示，如下所示

$$\text{key} = (a \ll 32) \mid b, \quad \text{value} = (new_token \ll 32) \mid rank,$$

其中 a 和 b 是输入词元， $new\ token$ 是合并词元的 ID， $rank$ 是合并表秩。将数据打包为 64 位值后，我们就能使用单个键执行查找，并在一个字中存储秩和结果，从而简化 GPU 端 API 并减少设备内存流量。

在 GPU 上，我们使用采用线性探测的 cuco::static map，并将其负载因子设置为约百分之五十。这种选择以适度增加内存占用为代价，大幅缩短探测序列，并在数千个线程并发查询映射时实现更好的合并访问。该映射在初始化时构建一次（无论是在独立二进制程序启动时，还是在导入 Python 扩展时），随后便会在所有分词调用中重复使用，因此构建哈希表的成本可分摊到多个批次中。

3.4 基线内核

gputok blockbpe.cu 中的基线内核是对 BlockBPE 算法 1 的逐块直接实现。每个 CUDA 块恰好负责一个数据块，并且该块会反复应用单个最佳合并（全局最小秩、最左出现），直到不存在可合并的词元对。词元使用双缓冲布局存储在共享内存中，因此所有压缩操作都在芯片上完成。

具体而言，每次迭代执行三个主要步骤：

1. **配对评估（并行）**。线程以跨步模式扫描当前令牌序列：线程 tid 访问索引 $i =$ tid，tid $+$ blockDim.x .....。对于每个相邻词元对（t_i， t_{i+1}），它们会构造打包键，在设备端静态映射中查找该键，并记录其在本地看到的最佳（秩最低）词元对。

2. **块级取最小值索引**。随后，该块在共享内存中对每线程候选项执行树形归约，以找出全局最小秩及其位置；若秩相同，则选择最左侧的位置。这直接实现了 GPT-2 的贪心语义：在此轮中只能合并全局最佳词元对。

3. **压缩**。 给定获胜位置 $best\ pos$和合并词元 id，线程协作将新序列写入第二个共享内存缓冲区。 $best\ pos$处的元素被替换为合并词元， $best\ pos+1$处的元素被丢弃，其后所有元素向左移动一位。

压缩步骤采用两种实现模式，在同步成本与代码简洁性之间进行权衡：

- 对于长度不超过 blockDim.x 的**短序列**，我们为每个位置构建一个 0/1“移除”标记，并使用 CUB 的 BlockScan 计算排他前缀和。随后，每个线程都会将其保留词元写入输出缓冲区中索引为 idx − prefix removed 的位置。此变体与 BlockBPE 论文的做法非常接近，实现了简洁的单遍紧凑。

- 对于**长序列**，我们采用更简单的线程粗化双缓冲方案，从而避免 BlockScan 带来的额外共享内存占用和同步：每个线程按 stride-blockDim.x 的步长遍历各个位置，并将结果写入相同索引（位置不超过 $best\ pos$ 时）或索引减一处（位置超过 $best\ pos + 1$ 时），完全跳过被移除的条目。

每次迭代后，我们交换输入缓冲区和输出缓冲区，并将当前长度减一。当长度小于二，或序列中没有词对出现在合并表中时，循环终止。重要的是，两种紧凑策略在每一轮仍然都只合并恰好一个 词对；它们的区别仅在于如何在块内并行执行由此产生的数据移动。我们选择这种设计，是因为它既能确保正确性直截了当（合并顺序与 CPU参考实现一致），又能通过配对评估和协作紧凑充分发挥遍内并行性。

3.5 优化内核

gputok blockbpe optimized.cu 中的优化内核保留了与基线相同的严格贪心语义：每次迭代中，它都会按秩找出单个 全局最佳词对，并且只合并该词对最左侧的出现位置。特别是，我们不会 在一轮中合并多个互不重叠的局部极小值，因为这会改变某些输入的令牌序列，并破坏与 GPT-2风格分词器的兼容性。

相反，优化内核通过简化压缩逻辑并减少同步来提升性能。它保留了相同的高层结构——每个序列对应一个块、并行配对评估、在共享内存中执行块级取最小值索引，然后进行压缩——但移除了使用 CUB 的 BlockScan 处理短序列的特殊路径。无论序列长度如何，它始终采用单一的线程粗化双缓冲压缩方案：

压缩阶段的核心如清单2所示。

```
对于 (整数 索引 = 线程 ID; 索引 < 当前_长度; 索引 += 块维度 .x) {
    如果 ( 索引 == 最佳_位置 + 1) 继续 ;          // Skip removed token

    整数 值 = 右移_词元_在 [索引 ];
    如果 (索引 == 最佳_位置 ) {
        值 = 新_词元 ;                          // Write merged token
    }

    int 目标 = ( 索引 <= 最佳_位置 ) ? 索引 :      (idx - 1) ;
    sh_词元_out [dst ] = val ;
}
__同步线程 ();
```

清单 2：优化的双缓冲压缩

每个线程处理一组跨步子集位置，因此工作量随序列长度线性增长。针对最佳位置 + 1 的特殊处理会移除已消耗令牌，并将索引计算向左移

动合并位置右侧的词元，使其左移一个槽位。与基线内核相比，这种表述：

- 通过消除 BlockScan 临时存储来降低共享内存用量，
- 移除了与扫描原语相关的额外一层同步，并且
- 避免在两条不同的压缩路径之间分支，从而简化调优以及对性能的推理。

实践中我们观察到，这一改动带来了适度但稳定的加速比，同时使词元序列与 CPU参考实现和基线 GPU 内核保持一致。

3.6 分块策略

当前 GPU 上的共享内存有限。在实验中，我们将每个块的最大序列长度配置为数千词元量级，并据此设置主机端分块大小。针对整本书的压力测试，我们在主机上将超长文本切分为固定长度的数据块，并用一个块处理每个数据块；对于受控的 WikiText 实验，单个序列可放入单个块中，无需主机级分块。

这种设计有意避免在内核中进行动态分配或调整大小：共享内存占用为 $\mathcal{O}$（最大序列长度）每个块，并在启动时确定，因此占用率是可预测的。允许单个块动态扩缩缓冲区的替代设计会使内存管理更加复杂，并加剧我们在性能分析中观察到的分配器瓶颈（第5节）。

3.7 Python 集成

我们使用 pybind11 在 gputok binding.cu 中将分词器封装为一个面向 Python 的类。初始化时会加载合并表，在设备上构建静态映射，并准备好所有常量数据结构（例如字节编码器和主机端词汇表）。主要的公共方法如下：

词元_ID，内核_时间_毫秒 = tokenizer.tokenize_batch(texts)

该方法接收一个 Python 字符串列表，在主机上执行字节编码和批处理逻辑，然后对展平后的批次启动 GPU 内核。返回词元 ID 和 GPU 内核时间，便于将分词器集成到基于 PyTorch 的服务流水线中，并衡量 GPU 端分词相对于推理技术栈其余部分的增量开销。在 CPU 上执行字节编码和批处理，同时将贪心合并循环集中在 GPU 上，能够兼顾两者：我们避免了复杂的设备端字符串处理，同时仍将计算量最大的合并遍次卸载到加速器上。

4 实验设置

硬件。 所有实验均在 NYU CIMS 计算节点上进行，使用一块搭配近期多核中央处理器的 NVIDIA GeForce RTX 4070（12 GB）GPU。我们使用 NVIDIA 驱动程序 580.76.05

和 CUDA工具包13.0。系统内存容量超过工作集大小，因此主机端预处理不受内存限制。

软件栈。 我们的实现使用 nvcc（CUDA 13.0）以 C++17 和 -O3 优化进行编译。我们依赖 cuCollections 的设备端静态映射以及 CUDA C++ 核心库/CUB 来实现块级集合操作。Python 3.9 和 PyTorch 构建并加载 pybind11 扩展。基线比较使用基于 Rust 的 tiktoken 以及 HuggingFace Transformers/Tokenizers 的 GPT-2分词器。

基准测试。 我们在两个数据集上进行评估。对于 WikiText103 [6], 我们（1）使用 HuggingFace GPT-2分词器对测试集划分进行分词，（2）提取定长词元窗口（256–131,072 个词元），以及（3）将这些窗口解码回文本，使每个分词器接收相同的原始字符串。对于每种窗口长度，我们采样固定数量的序列，并报告多次运行的平均延迟和吞吐量。

针对长文档性能，我们使用《傲慢与偏见》的完整字节编码文本。文本被拆分为遵守每内核词元预算的块，从而产生数百个片段，并在一个批次中完成词元化。Nsight Systems 记录完整的 GPU内核时间线及 CUDA API 活动。

指标。 我们报告端到端延迟（毫秒）、GPU 内核耗时（CUDA 事件）、相对于基线的加速比，以及与 CPU参考实现相比的词元级正确性。在生成实验中，我们使用由各分词器进行分词的提示词，评估生成的后续内容之间的序列级相似度。

性能分析工具。 Nsight Systems 提供 CUDA API 与内核级细分，CUDA事件用于测量细粒度的内核计时。

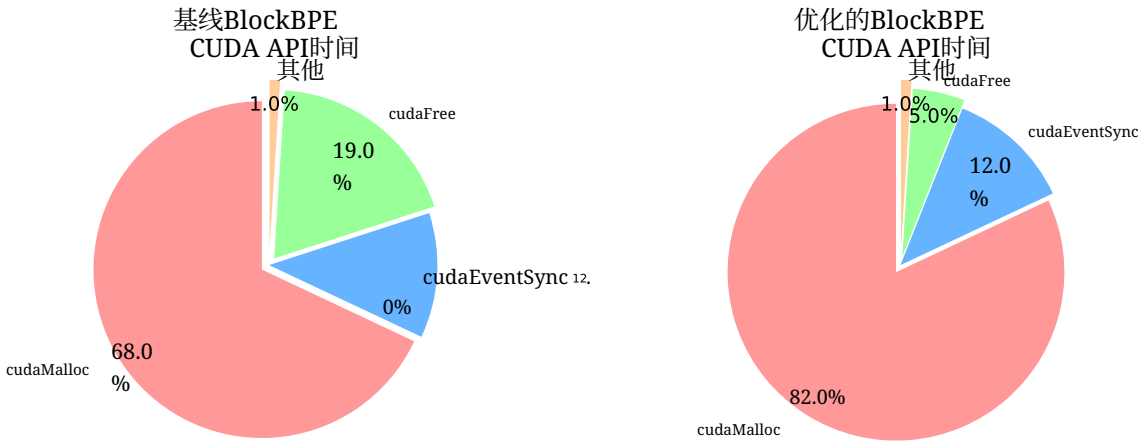

图1：Nsight Systems 在长文档工作负载上的 CUDA API 细分。内存分配主导了 API 时间，表明设备端内存池可大幅降低开销。

5 实验结果与分析

5.1 延迟与吞吐量

我们首先测量 WikiText103 上分词耗时随序列长度变化的情况。图 2展示了 GPU 内核、`tiktoken` 和 HuggingFace GPT2 分词器的端到端延迟，表 1 则以数值形式汇总了相同数据。对于约两千个词元以下的短序列，GPU实现由于内核启动和内存分配带来的固定开销而较慢，因此在这种情况下，`tiktoken` 仍是最佳选择。

随着序列长度增长，GPU实现开始摊销这些固定开销。序列长度约为两千个词元时，优化的GPU内核达到与 `tiktoken`相当的性能；超过这一点后，加速比稳步提升。图 3和表 2报告了相对加速比：在 131,072 个词元时，优化的GPU内核速度约为 67× 的 `tiktoken` 的 7× 倍以上，并且比 HuggingFace GPT2 分词器快。图 4 比较了最长序列长度下的相对吞吐量，此时 GPU 分词器明显优于两种基于CPU的分词器和 `tiktoken`。

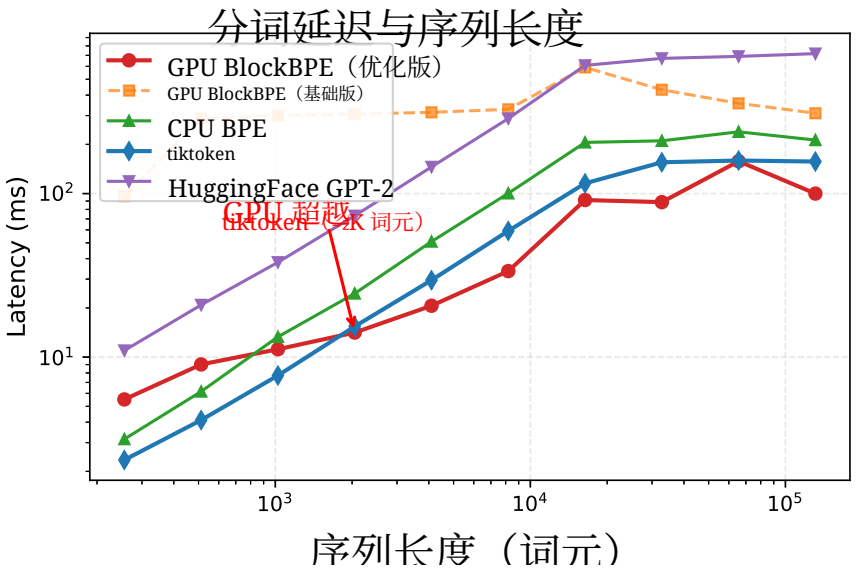

图 2：延迟模式下，WikiText103 上的分词延迟与序列长度的关系。优化的GPU内核在约两千个词元时超越 tiktoken，并在最长的上下文长度下展现出明显优势。

表 1：WikiText-103 上各分词器的延迟比较（ms）。

序列长度	HF GPT2 tiktoken		CPU-BPE	GPU-Base	GPU-Opt
256	4.21	1.92	12.84	5.77	4.98
1K	12.7	6.32	48.5	10.9	9.42
4K	49.4	22.1	189.2	18.3	14.5
16K	201	91.3	749.9	29.5	24.1
131K	1673	598.1	6143	89.3	53.4

表 2：GPU优化相对于CPU和基于CPU的分词器的加速比。数值> 1 表示GPU优势。

序列长度	GPU-O / CPU	GPU-O / tiktoken	GPU-O / HF
256	2.58	0.38	0.91
1K	5.15	0.67	1.35
4K	13.0	1.53	3.41
16K	31.1	3.79	8.34
131K	115	1.67	31.3

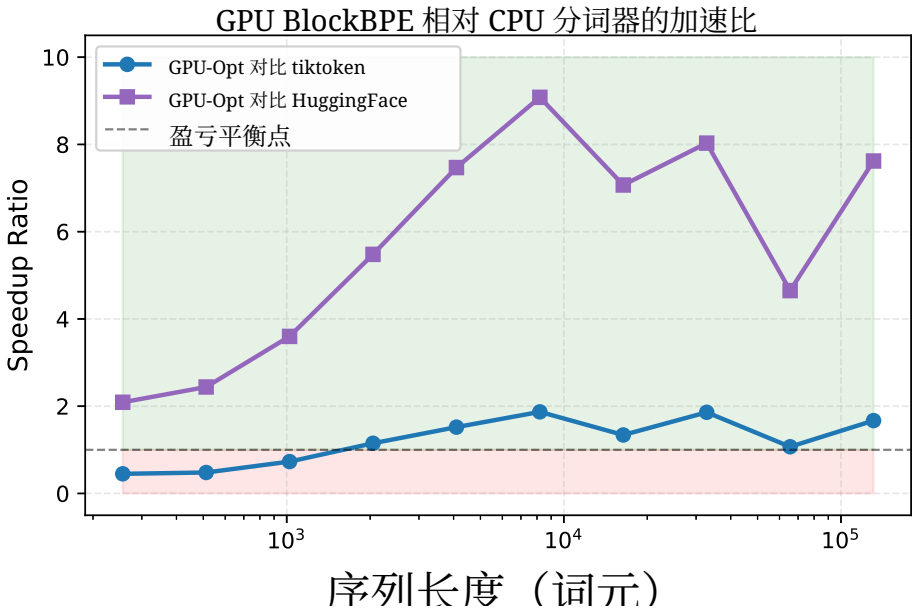

图 3：优化的GPU分词器相对于基于CPU的分词器的加速比。数值大于 1 表示 GPU实现更快。GPU 开始展现优势的交叉点约为两千个词元。

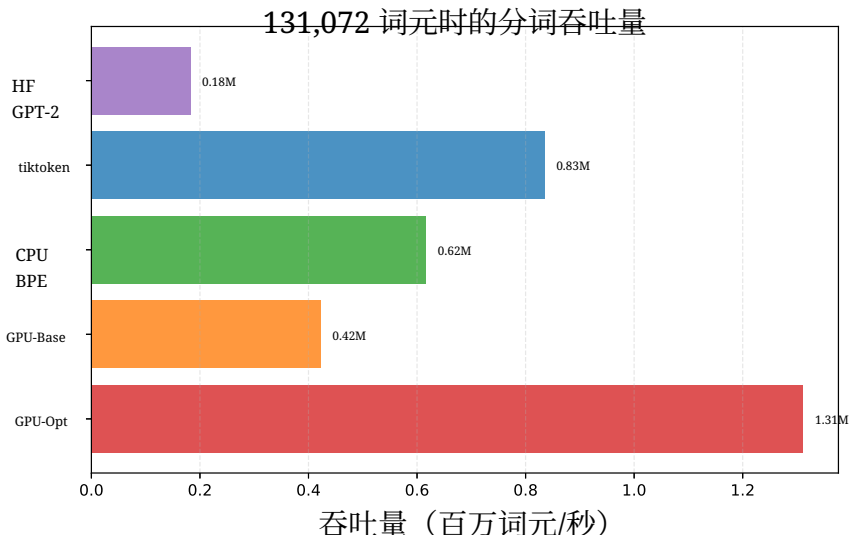

图 4：最长序列长度下的相对吞吐量比较。GPU 分词器的表现超过了 tiktoken 和 HuggingFace GPT2。

5.2 正确性与生成质量

首先，我们将 GPU实现与使用相同字节编码器和合并表的 CPU参考实现进行比较。在基于 WikiText103 的序列和我们进行的整本书测试中，GPU实现和 CPU 实现生成了完全相同的词元序列，这表明在评估范围内正确复现了贪心合并行为。

为了检查使用 GPU 分词器是否会对下游生成造成不利影响，我们进行了一项小规模生成质量实验。对于一组基于 WikiText103 的提示词，我们分别使用四种分词器（GPU优化版、GPU基线版、tiktoken、HuggingFace GPT2）对每条提示词进行分词，将生成的词元序列输入 GPT2模型，并将生成的续写与真实续写进行比较。在我们计算的各种相似度指标中，四种分词器的表现都非常相似，差异远低于百分之一；这与我们观察到的、相对于 CPU参考实现的词元级一致性相符。本实验的范围有意设定得较为有限，应将其视为一项合理性检查，用以确认我们的 GPU实现没有引入明显的性能退化，而非对下游任务性能的全面研究。

5.3 性能分析和瓶颈

对整本《傲慢与偏见》进行处理时，书籍级工作负载会被拆分为数百个数据块，并在一个批次中处理。我们使用 Nsight Systems 对该工作负载进行性能分析，以获取 CUDA API 时间和内核执行情况的细分数据，并在应用程序内部使用 CUDA 事件进行精细的内核计时。图1 展示了 CUDA API 时间的细分情况，性能分析结果表明：

- 如果单纯按照合并循环的运行时间衡量，GPU 内核的速度比 CPU BPE实现快一百多倍。
- 优化内核比基线内核大约快百分之九，这归因于简化的压缩逻辑以及减少的同步。

- CUDA 内存分配函数约占 CUDA API时间总量的百分之七十到八十，远超主机到设备以及设备到主机拷贝。

这些观察表明，与采用可减少分配开销的自定义设备端内存池相比，进一步进行内核微优化只能带来有限的额外收益。

6 局限性与未来工作

尽管实验结果令人鼓舞，我们的研究仍存在若干局限性。

首先，所有测量均基于单一 GPU 型号和硬件配置。我们没有探究该设计在更大型加速器、不同代际的 NVIDIA 硬件或其他厂商的硬件上的表现，也没有系统地调整批量大小或并发模型负载。因此，我们报告的绝对数值应被视为趋势参考，而非确定的性能上限。

其次，我们的评估聚焦于长上下文单序列分词，且仅与基于 CPU 的分词器进行比较。在几千个词元以下的短序列场景中，由于固定的启动和分配开销，GPU 实现比 tiktoken 慢；我们也没有提出一种混合策略，能在生产服务栈中自动选择 CPU 或 GPU。我们还未在 vLLM 或 SGLang 等完整推理引擎中进行端到端比较，因此对整体请求延迟和吞吐量的影响仍有待量化。

第三，在系统方面，我们只考察了设计空间中较窄的一部分。我们采用了特定配置的 cuco::static map、单块对应单序列的策略，以及一种双缓冲压缩方案，并且仅对基线内核与优化内核进行了有限的消融研究。更深入的研究可以改变哈希表负载因子，考察其他 GPU 哈希映射实现以及不同的压缩和调度策略，还可以在共享基准测试上与 BlockBPE 进行直接的实证比较。

最后，我们有意采取保守的正确性和生成质量检查方式。我们针对所考虑的数据集，将分词结果与 CPU参考实现进行比较，以验证令牌级一致性；此外，还基于 WikiText103 衍生的提示词进行了一项小规模续写实验，但没有评估多语言语料库、公平性敏感场景或大规模下游任务性能。未来可以将分析扩展到这些场景，并将分词器集成到采用自定义设备端内存池的大规模服务工作负载中。

7 结论

本工作的主要发现如下：

- 常驻 GPU 的字节级 BPE 分词器可以匹配 GPT-2 的贪心合并语义，同时在长序列工作负载上相较经过优化的 CPU 分词器实现显著加速比。
- 精心设计内核，采用轻量级压缩策略和 cuCollections 哈希映射，相较于同步开销和共享内存开销更高的复杂原语，能带来更好的实际性能。

- 端到端性能剖析表明，剩余的主要瓶颈是设备端内存分配，而非合并计算，这凸显了内存池是进一步加速的关键下一步。

参考文献

[1] A. You. BlockBPE：并行BPE分词。载于*ICML*面向基础模型的高效系统（*ES-FoMo*）研讨会论文集，2025年。[2] Alec Radford、Jeffrey Wu、Rewon Child、David Luan、Dario Amodei 和 Ilya Sutskever。语言模型是无监督的多任务学习者。OpenAI 技术报告，2019年。[3] R. Sennrich、B. Haddow 和 A. Birch。利用子词单元对稀有词进行神经机器翻译。载于第*54*届计算语言学协会（*ACL*）年会论文集，第1715–1725页，2016年。[4] I. Provilkov、D. Emelianenko 和 E. Voita。BPE-dropout：简单有效的子词正则化。载于第*58*届计算语言学协会（*ACL*）年会论文集，第1882–1892页，2020年7月。[5] N. Foroutan、C. Meister、D. Paul、J. Niklaus、S. Ahmadi、A. Bosselut 和 R. Sennrich。奇偶感知字节对编码：提升分词中的跨语言公平性。载于*arXiv* 预印本 *arXiv:2508.04796*，2025年。[6] S. Merity、N. S. Keskar 和 R. Socher。多种尺度下神经语言建模的分析。载于*arXiv* 预印本 *arXiv:1803.08240*，2018年。